UNIX
Questions and Answers

Suresh Basandra

UNIX Questions and Answers

PDF ISBN-10: 0983592837
PDF ISBN-13: 9780983592839
EPUB ISBN-13: 9781452487687

Current Edition: 2011
First Edition: September 2010

Published and typeset by Basandra Books, California, USA
Designer: Karuna Basandra
Webmaster: Gagan Basandra
Editor: Kriti Basandra
Visit Basandra Books on the Web: http://books.basandra.com/
Preface

A few years ago we could basically pick and choose the job we wanted. If we were dissatisfied we moved on. Job mobility was the order of the day. Now, in far too many cases, outsourcing, downsizing, and/or rightsizing seem to be the order of the day. Job stability often appears to be little more than words found in a dictionary.

Here is a book that will enable you to beat out the competition. Your competition is all those people trying to get the same job you are trying to get. Let us face it, as far as you are concerned there is only one person that is going to get each job. As for the job you want, you want it to be you, not them. If you play your cards right, try as hard as you can, prepare yourself, and never give up, you will succeed. And this is what this book is all about … preparing and succeeding.

There are numerous FAQs, questions and answers, articles on Web, books, etc. on the market covering almost every conceivable topic, however, from an interview perspective it is not feasible to brush up on all these materials when you may not have much time. With technologies evolving so rapidly, it is hard for even the most knowledgeable professional to handle job interview with confidence and precise answers to tough technical and management questions. This is especially true when it comes to information about such vital new areas of job opportunity as UNIX, Networking, etc. Here is a book that will enable you to ace any technical interview as it provides answers to challenging and difficult questions in a wide range of important areas. The key objective of this book is to cover all the core concepts and areas which all UNIX developers, designers, administrators, and architects should be familiar with to perform in their jobs or to do well in interviews. The interviewer can also use this book to ensure that they hire the right candidate depending on their requirements.

UNIX Questions and Answers is a one-stop reference that both beginning and experienced software engineers will use successfully in technical job interviews and countless on-the-job situations. Providing the answers to critical questions, from the simplest to the most advanced, this book is arranged to get you the information you need the moment you need it. You will find helpful explanations of crucial UNIX issues.

Packed with numerous real-life examples and case studies, UNIX Questions and Answers is an indispensable guide covering all the areas of UNIX. It is a practical and constantly usable resource for understanding fundamental UNIX issues and implementing workable solutions based on author's more than 30 years development and management experience.

There are in all about 300 questions with detailed answers so you do not have to worry about spending countless hours browsing the Web or text books for correct answers that get you succeed. This book is not a multiple choice question and answer book as we believe we need to have detailed understanding of concepts and principles to better do our job as a software engineer, administrator, or architect. This book also does not contain any non-relevant material to avoid making it bulky.

Key topics covered include: UNIX Basics, UNIX Kernel, Utilities, File Structure and Directories, Shell Fundamentals, Shell Commands, Shell Programming, Shell Scripting, Threads and Processes, Commands, Networking, Sockets, Protocols, LAN, TCP/IP, System Administration, Network Administration, Managing UNIX Jobs, User Environment, Performance Tuning, Installation and Configuration, Glossary, Acronyms, Visual Editor, Makefile, sed, awk, Regular Expressions, Mail, Linux, etc.

In the latest edition, we have demonstrated a commitment to clear, direct writing, with questions that are quick, to the point and stress the core essentials of UNIX. Those who want to learn more about the profession, as well as those who want to fine-tune their development skills, will find it to be a straightforward question and answer format with hundreds of questions covering key UNIX knowledge areas. Whether you are a seasoned professional, novice, student, or instructor of UNIX, you will appreciate this book for its rich content and ability to test your skill and knowledge.

Simply return the book if you are not 100% satisfied - no questions asked. However, I would request you to please provide your valuable suggestions for improvement since I strongly believe in continuous improvement.

More than 200,000 copies of author's previous books have been sold. Please review author's profile at: http://books.basandra.com/aboutus.html

It is planned to keep this book updated on a regular basis so that the readers are kept current with latest trends.

Basandra Books introduced its first question and answer book in 2010 and, as the software engineering profession continues to mature, improvements to this book have evolved naturally from various feedbacks on the previous editions. I sincerely appreciate the invaluable feedback provided by various readers and reviewers. Please provide any feedback and corrections using http://books.basandra.com/contactus.html or email to books-support@basandra.com. Thank you in advance for your feedback.

All in all a lot of effort by the author has been put into providing you, the technical and management professional whose livelihood depends on the currency of his or her technical and management knowledge and skills, with as much new meaningful information as possible. If I am able to give you any of the information that you need to survive and succeed then my effort would be considered more than worthwhile.

Suresh Basandra,
California, USA

1.11. What are the disadvantages of using threads? Discuss some problems that can result from incorrect thread programming.
1.12. What is deadlock? How can I eliminate it?
1.13. What is the difference between multithreading and multitasking? What about multiprogramming? Multiprocessing?
1.14. Describe program, process, fork, exec, waitpid.
1.15. What is a fork in UNIX?
1.16. What are the problems with fork? How threads do solves these problems?
1.17. What is a daemon?
1.18. What is a cron job?
1.19. What is a socket?
1.20. What is the main advantage of creating links to a file instead of copies of the file?
1.21. Describe hard link.
1.22. Describe soft link.
1.23. What is the difference between a symbolic link and a hard link?
1.24. Write a command to find all of the files which have been accessed within the last 30 days.
1.25. What is the most graceful way to get to run level single user mode?
1.26. What does the following command line produce? Explain each aspect of this line. $ (date ; ps -ef | awk {print $1}' | sort | uniq | wc -l) >> Activity.log
1.27. In UNIX OS, what is the file server?
1.28. What is NFS? What is its job?
1.29. What is CVS? List some useful CVS commands.
1.30. What do these permissions on a file mean? =-rwxr-xr-x
1.31. What is a tar file?
1.32. Describe gzip.
1.33. How do files differ in Windows, UNIX and Mac?
1.34. What is the difference between .so and .a files in UNIX?
1.35. Windows vs. UNIX: Linking dynamic load modules
1.36. How would you find all the processes running on your computer?
1.37. List some UNIX commands and their use.

1.71. What is the default windowing system used in UNIX?
1.72. What command is used to list the contents of a directory?
1.73. What command is used to display a list of currently running processes?
1.74. What is a login shell?
1.75. What is a UID?
1.76. In what file is the relationship of UID to username defined?
1.77. What command is used to check a filesystem for errors?
1.78. Is there a difference between a file called OUTPUT.TXT (in all caps) and output.txt (all lowercase) in UNIX?
1.79. What command is used to read the manual page for a given command?
1.80. What symbol is used to redirect standard out (STDOUT) to a file?
1.81. What command is used to establish a secure command-line session with another system over a TCP/IP network?
1.82. What file contains the list of drives that are mounted at boot?
1.83. In what file is the default runlevel defined?
1.84. Define and describe TCP Wrappers. What are the reasons for using TCP Wrappers in addition to a host-based firewall?
1.85. Define and discuss the security ramifications of each of these situations: Current directory (.) in root's $PATH variable, r commands (rlogin, rcp, etc), setUID bit on /bin/vi
1.86. What is the advantage of giving a user elevated privileges to certain commands through sudo, as opposed to giving them root access to a system?
1.87. What is the chroot command and why is it important to the security of a system?
1.88. Describe a buffer overflow attack and list possible defenses.
1.89. What is the weakest point in any network or system's security? What can you do about it?
1.90. Describe a file system integrity tool and explain its importance.
1.91. What is an intrusion detection system and what are some basic flaws in the concept of 'detecting intrusion'?
1.92. What is ARP cache poisoning, and how can it be prevented?

1.93. Describe VPN technology and list some common implementations.
1.94. What is a DMZ and why would you want one?
1.95. On Linux, how can the directory entries at the top of the /proc filesystem aid in security?
1.96. What command is used to display the ports that are open on a UNIX system?
1.97. What is the subnet mask for a default Class-C IP network?
1.98. In a standard class C IP network, how many IP addresses can be assigned to hosts? What are the other non-assignable addresses used for?
1.99. What is a MAC address?
1.100. You wish to set up a Linux system as a router between two subnets. You have installed and configured two network cards, each attached to a different subnet. What are the steps that need to occur before the system will act as a router?
1.101. What is an MTU?
1.102. What file determines the DNS servers that are queried when a network request is made?
1.103. What is the difference between UDP and TCP?
1.104. What is the difference between POP and IMAP?
1.105. How do I know if I should use IMAP or POP?
1.106. Describe Simple Mail Transfer Protocol.
1.107. What steps are necessary to configure a Solaris system to participate in an IP network?
1.108. What is TCP sequence number prediction and what can you do to prevent it?
1.109. You have purchased a domain name and setup one of your UNIX servers as a DNS server. What entry must you make in your primary zone file to ensure that email addressed to your domain is sent to the correct mail server?
1.110. Associate these common network services with their default port numbers: Telnet, SSH, FTP, SMTP, DNS, POP3, HTTP, HTTPS, IMAP
1.111. What command will display the routing table on a UNIX system?

1.112. Your UNIX system is able to ping other devices on the local subnet, but it is unable to ping a device outside of the subnet. What could be the problem?

1.113. When you initiate a telnet session to your UNIX system, the connection is established, but there is a 20 second or more pause before you see a login prompt. What could cause this?

1.114. Describe the default partitioning scheme in Solaris, denoting the slice number, the default mount point, and the purpose.

1.115. Define and describe the following terms related to disk management under UNIX: Boot block, Super block, Logical block, Inode

1.116. Describe the processes for adding a new hard disk to a UNIX system.

1.117. What is the difference between a regular file system and a journaling file system?

1.118. Define and describe the common UNIX file systems.

1.119. What are the 3 settings in the standard UNIX file permissions model, and how does each apply to files and directories?

1.120. What is the setUID bit and how is it employed?

1.121. Define and describe the cron job scheduling system. What are the fields available to define the time a job will execute?

1.122. In the standard lpr-based UNIX printing system, what command will display the status of a print queue?

1.123. Describe the Samba suite of utilities. How are they useful in a heterogeneous environment?

1.124. Describe the package management options available on Linux systems.

1.125. Describe the package management software on Solaris.

1.126. What are three ways to find a file on a UNIX system?

1.127. If /etc/inittab defines your default runlevel as 3, where do your system startup scripts reside?

1.128. You are logged in as root and wish to kill a process with PID #1054. You type ‘kill 1054’ but you still see the process in the process table. Why did the process not die, and what can you do to get rid of it?

1.129. What are the fields defined in the /etc/passwd file?

1.130. What is the purpose of the ‘grep’ command?

1.131. What command is used to display information about the network interfaces on a system?

1.132. The 'ps' command lists processes currently running on the system. Identify and describe the following column headers from the output of 'ps -aef': UID, PID, PPID, TTY

1.133. In Solaris, what command is used to display the slice information for a particular disk?

1.134. What command is used to display the disk space occupied by mounted file systems?

1.135. What file contains a list of currently mounted file systems?

1.136. On Solaris, what file contains settings that define the default maximum time a password is valid and the default minimum time period before a password must be changed?

1.137. What command and options would you issue to set the permissions on a file to read, write, execute for owner, read and execute for group, and no permissions for other?

1.138. What command will display the default file permissions that will be assigned when a user creates a file?

1.139. On Solaris, what command is used to display the Access Control List entries for a file?

1.140. What command is used to run an application with a different security context than the user's default? What is the configuration file for this command, where programs are listed which defined users can execute with higher privileges?

1.141. What file defines the default runlevel of the system and dictates what directory holds the startup scripts for that runlevel?

1.142. What command is used to check a filesystem for errors?

1.143. Why are there normally two bin directories, /bin and /usr/bin?

1.144. On Solaris, what file defines the master kernel configuration file?

1.145. At the Solaris OpenBoot prompt, what command will display all of the SCSI devices connected to the system?

1.146. How do you boot a Solaris system into single user mode?

1.147. What command allows you display and to modify your keyboard mappings in X Windows?

1.148. In Linux, what file contains a list of all IRQs currently in use?

1.149. You attempt to unmount a currently mounted disk and you receive the error Unable to umount: device or resource busy. What does this mean and what must you do before you can unmount the disk?

1.150. You need to run the fsck program against a file system, but you must do this while the file system is unmounted. Since the file system in question is the root file system, you cannot unmount it while the system is running. What options do you have for running fsck against this file system?

1.151. You have forgotten root's password. What options do you have to get back into the system as root?

1.152. Your UNIX workstation is unable to ping any hosts on the local TCP/IP network. What steps do you take to troubleshoot this problem?

1.153. Your production UNIX system has been running fine when users suddenly complain that the system is slow. You log in and the system feels 'sluggish' and the terminal is unable to keep up with your typing. What steps do you take to diagnose the problem?

1.154. Describe the steps the system goes through, at boot time, before the users may sign on.

1.155. When, and depending on what, are the other filesystems checked and mounted?

1.156. Name two ways to bring the system down.

1.157. What are the files checked or scripts executed when a user logs in?

1.158. When an account, for various reasons, is not used anymore, how can the system administrator disable it?

1.159. How can a user ensure that certain sensitive files cannot be read by others?

1.160. How can a user remind another user about an important meeting taking place on a certain date, at a certain hour?

1.161. How can a user execute a long command and still have the terminal available for work?

1.162. Once the background command has been started, can it be aborted like any other foreground command?

1.163. What is a filesystem?

1.164. What happens if you have a directory such as /tmp, and you mount a filesystem such as /dev/u, on this directory?

1.165. Can the system administrator enable users to mount a filesystem?

1.166. Does it make sense for a directory to have the execute permission on it?

1.167. By default, the GID (group identifier) of a newly created file is set to the GID of the creating process or user. How can this behavior be changed?

1.168. What is a link between files, and what is the command you would use to link files?

1.169. What are the two types of links?

1.170. Which type of link consumes an inode, and which type does not?

1.171. What is the most common way of locating files?

1.172. How can you search a file for occurrences of a word or phrase?

1.173. How can you list the names of all files in all subdirectories in which there is a certain pattern?

1.174. Assuming you have a big file and you want to clear it (empty it), but you want to keep the file and its permissions, what could you do?

1.175. Assuming that a directory becomes too large (has too many entries) and you decide to make two directories out of it (one of which has the same name), how would you proceed?

1.176. What is the purpose of the lost+found directory in a filesystem?

1.177. List three commands most often used for archive or data transfer between devices.

1.178. Once you obtain a stable system, that is, a good kernel, what are some of the first things the system administrator should do?

1.179. What steps should be followed when you find you have an unusable root filesystem?

1.180. Can you copy files to a DOS floppy on a UNIX system?

1.181. How can you find out the baud rate, the parity scheme, and other information about a serial line?

1.182. How do you set the terminal type?

1.183. What performance tools can you use to diagnose system inefficiency?
1.184. How can the time command be used?
1.185. What is swapping (paging)?
1.186. What should you do in the case of an intense swapping/paging activity?
1.187. How can you check the status of processes?
1.188. How can you stop a runaway process?
1.189. Assuming that you have to start the execution of a long command and you want to log out and go home, how do you go about it?
1.190. How can you prioritize processes?
1.191. How can you schedule programs to run at specific times?
1.192. What is a shell?
1.193. How does the shell execute the commands?
1.194. How does the shell locate commands that the user wishes to execute?
1.195. Which are the characters you could use to form regular expressions to match other characters?
1.196. How can you redirect the output of a command?
1.197. How can a command take the input from a file?
1.198. What is command substitution?
1.199. What are positional parameters (arguments)?
1.200. What is a pipeline for a shell script?
1.201. How can you use parentheses or braces at the command line or in a shell procedure?
1.202. What is the difference between using parentheses and using braces?
1.203. What are functions from the shell's standpoint?
1.204. What is a command's environment?
1.205. How can you initialize new variables in a shell script and use them later in the sign-on shell?
1.206. How can you solve the following situation: You want the users, once signed on, to be put in the application program; once they quit the application, you want them signed off the system?
1.207. How can you make sure that the work file created during the execution of a script will be removed even if the script was interrupted?

1.208. Assuming you have to run backups during the night, and there is no night operator and the backup utility prompts you for answers before it starts, how would you solve this problem?
1.209. How can you write the standard output and the standard error for a command to the same file?
1.210. In a command list, how can you arrange the commands, so if one of them fails, the execution of the list stops?
1.211. In a command list, how can you arrange the commands so that, the second one is executed only if the first one fails?
1.212. What can you do to fix the problem if one of your filesystems runs out of inodes?
1.213. How can you check the amount of free space on filesystems?
1.214. What does it mean if a user has write permissions on a directory but is unable to remove files from that directory?
1.215. What command can be used to compare two text files?
1.216. How can you convert a text file from UNIX format to MS-DOS format?
1.217. What is umask?
1.218. How can you find out what variables are currently assigned?
1.219. What command can you use to archive a raw device?
1.220. What are device drivers?
1.221. What are UNIX daemons?
1.222. Assuming you want to start the database engine every time you boot the system, how would you do it?
1.223. When the system is coming down, how would you bring down the database engine?
1.224. What can you do if a scrambled terminal responds to keyboard input but the display is incorrect?
1.225. What can you do if a terminal that responds to keyboard input does not display the characters entered at the keyboard?
1.226. How can all the files ending in .foo in a directory be renamed to end in .bar?
1.227. Why wouldn't the administrator be able to unmount a filesystem?

1.250. How do I tell inside .cshrc if I'm a login shell?
1.251. How do I find the last argument in a Bourne shell script?
1.252. What's wrong with having '.' in your $PATH?
1.253. What is the significance of the tee command?
1.254. What does the command $who | sort –logfile > newfile do?
1.255. What does the command $ls | wc –l > file1 do?
1.256. Which of the following commands is not a filter (a) man, (b) cat, (c) pg, (d) head
1.257. How is the command $cat file2 different from $cat >file2 and >> redirection operators?
1.258. Explain the steps that a shell follows while processing a command.
1.259. What is difference between cmp and diff commands?
1.260. What is the use of 'grep' command?
1.261. What is the difference between cat and more command?
1.262. Write a command to kill the last background job?
1.263. Which command is used to delete all files in the current directory and all its sub-directories?
1.264. Write a command to display a file's contents in various formats?
1.265. What will the following command do?
1.266. Is it possible to create a new file system in UNIX?
1.267. Is it possible to restrict incoming message?
1.268. What is the use of the command ls -x chapter[1-5]
1.269. Is 'du' a command? If so, what is its use?
1.270. Is it possible to count number of char, line in a file; if so, How?
1.271. Name the data structure used to maintain file identification?
1.272. How many prompts are available in a UNIX system?
1.273. How does the kernel differentiate device files and ordinary files?
1.274. How to switch to a super user status to gain privileges?
1.275. What are shell variables?
1.276. What is redirection?
1.277. How to terminate a process which is running and the specialty on command kill 0?

1. UNIX

1.1. Windows NT is a 32-bit operating system (OS). What does it mean to be a 32-bit OS?

A 32-bit OS runs on at least a 32-bit processor and generally makes a flat, 32-bit virtual address space available to programs. Programs running on a 32-bit OS will generally use 32 bits as their fundamental word size. For instance, an integer will be a 32-bit value rather than a 16-bit value. The term "32-bit OS" has also come to imply a variety of technologies, such as preemptive multitasking and process isolation that are common to most 32-bit operating systems but not necessarily inherent to a 32-bit operating system.

1.2. Describe programs, processes, and threads.

A program is a collection of instructions and data that is kept in a regular file on disk. In its i-node the file is marked executable, and the file's contents are arranged according to rules established by the kernel. This is another case of the kernel caring about the contents of a file.

Programmers can create executable files any way they choose. As long as the contents obey the rules and the file is marked executable, the program can be run. In practice, it usually goes like this: First, the source program, in some programming language (C or C++, say), is typed into a regular file, often referred to as a text file, because it is arranged into text lines. Next, another regular file, called an object file, is created that contains the machine-language translation of the source program. This job is done by a compiler or assembler (which are themselves programs). If this object file is complete (no missing subroutines), it is marked executable and may be run as is. If not, the linker (sometimes called a "loader" in UNIX jargon) is used to bind this object file with others previously created, possibly taken from

collections of object files called libraries. Unless the linker could not find something it was looking for, its output is complete and executable.

In order to run a program, the kernel is first asked to create a new process, which is an environment in which a program executes. A process consists of three segments: instruction segment, user data segment, and system data segment. In UNIX jargon, the instruction segment is called the "text segment." The program is used to initialize the instructions and user data. After this initialization, the process begins to deviate from the program it is running. Although modern programmers don't normally modify instructions, the data does get modified. In addition, the process may acquire resources (more memory, open files, etc.) not present in the program.

While the process is running, the kernel keeps track of its threads, each of which is a separate flow of control through the instructions, all potentially reading and writing the same parts of the process's data. Each thread has its own stack, however. When you are programming, you start with one thread, and that is all you get unless you execute a special system call to create another. So, beginners can think of a process as being single-threaded. Not every version of UNIX supports multiple threads. They are part of an optional feature called POSIX Threads, or "pthreads."

Several concurrently running processes can be initialized from the same program. There is no functional relationship, however, between these processes. The kernel might be able to save memory by arranging for such processes to share instruction segments, but the processes involved can't detect such sharing. By contrast, there is a strong functional relationship between threads in the same process.

A process's system data includes attributes such as current directory, open file descriptors, accumulated CPU time, and so on. A process cannot access or modify its system data directly, since it is outside of its address space. Instead, there are various system calls to access or modify attributes.

A process is created by the kernel on behalf of a currently executing process, which becomes the parent of the new child process. The child

inherits most of the parent's system-data attributes. For example, if the parent has any files open, the child will have them open too. Heredity of this sort is absolutely fundamental to the operation of UNIX. This is different from a thread creating a new thread. Threads in the same process are equal in most respects, and there is no inheritance. All threads have equal access to all data and resources, not copies of them.

An application consists of one or more processes. A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. A thread is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a base priority, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Microsoft Windows supports preemptive multitasking, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

A job object allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

1.3. What is the difference between a thread and a process?

A process is an OS-level task or service. A thread runs "inside" a process and may be virtual or simulated. Generally speaking, threads share resources like memory, where processes each have their own separate memory area, and need to take more elaborate steps to share resources. Another name for thread is "lightweight process" to distinguish it from the "heavyweight" system processes.

In a nutshell, a process can contain multiple threads. In most multithreading operating systems, a process gets its own memory address space; a thread does not. Threads typically share the heap belonging to their parent process. Even though they share a common heap, threads have their own stack space. This is how one thread's invocation of a method is kept separate from another's. For instance, a JVM (Java Virtual Machine) runs in a single process in the host OS. Threads in the JVM share the heap belonging to that process; that is why several threads may access the same object. This is all a gross oversimplification, but it is accurate enough at a high level. Lots of details differ between operating systems.

Process is a execution of a program and program contains set of instructions but thread is a single sequence stream within the process. Single thread allows a OS to perform single task at a time. Thread is nothing but flow of execution whereas process is nothing but a group of instructions which is similar to that of a program except which may be stopped and started by the OS itself.

Similarities between process and threads are:

- Share CPU
- Sequential execution
- Create child
- If one thread is blocked then the next will be started to run like process.

Dissimilarities:

- Threads are not independent like process.
- All threads can access every address in the task unlike process.
- Threads are designed to assist one another and process might or might not be assisted on one another.

1.4. What is the difference between a lightweight and a heavyweight process?

Lightweight and heavyweight processes refer the mechanics of a multi-processing system.

In a lightweight process, threads are used to divide the workload. Here you would see one process executing in the OS (for this application or service.) This process would possess one or more threads. Each of the threads in this process shares the same address space. Because threads share their address space, communication between the threads is simple and efficient. Each thread could be compared to a process in a heavyweight scenario.

In a heavyweight process, new processes are created to perform the work in parallel. Here (for the same application or service), you would see multiple processes running. Each heavyweight process contains its own address space. Communication between these processes would involve additional communications mechanisms such as sockets or pipes.

The benefits of a lightweight process come from the conservation of resources. Since threads use the same code section, data section and OS resources, less overall resources are used.

The drawback is now you have to ensure your system is thread-safe. You have to make sure the threads do not step on each other. Fortunately, C++ or Java provides the necessary tools to allow you to do this.

1.5. What is a fiber?

A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

1.6. What is multitasking?

A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor time slice to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors. However, you must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

1.7. Describe synchronization with respect to multithreading.

With respect to multithreading, synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared variable while another thread is in the process of using or updating the same shared variable. This usually leads to significant errors.

1.8. Why is it sometimes necessary to synchronize the actions of multiple threads?

It is sometimes necessary to synchronize the actions of multiple threads to maintain read or write consistency between shared resources.

For example, say, account balance is $500.
Without synchronize

Assume two threads.
Thread one reads the value = $500
Thread two changes it to = $450
It gets blocked by CPU
Thread two reads the value = $500. It should have got the value $450.

Java implements this by monitors. C and C++ use semaphores.

1.9. What are mutex and semaphore? What is the difference between them?

A mutex is a synchronization object that allows only one process or thread to access a critical code block. A semaphore on the other hand allows one or more processes or threads to access a critical code block. A semaphore is a multiple mutex.

1.10. How do you make methods thread-safe?

It is very difficult to prove that an object is thread-safe. The main rule of thumb for making thread-safe objects is, "Make all the instance variables private, and all the public accessor methods synchronized." However, this is sometimes difficult to achieve in practice, due to exigencies of performance, architecture, or implementation.

1.11. What are the disadvantages of using threads? Discuss some problems that can result from incorrect thread programming.

Some problems resulting from incorrect thread programming are race conditions, deadlock, livelock, starvation, busy waiting, and over-locking.

Race: A race condition happens when threads are not correctly synchronized. Race conditions often occur when two threads are accessing and updating a global variable at the same time. Deficient locking of shared resources is one of the causes of races. Races cause unpredictable results and may cause quite incorrect behavior. However, races sometimes don't show up until the JVM, computer or other thing changes so as to affect the timing of the program execution. For example, consider the case where two threads simply increment a global variable that initially has the value 5. Incrementing often requires three operations in machine code: a load, an

addition, and a store. If the first thread does a load, obtains the value 5, and then gets swapped out, the second thread might also do the load, obtain the value 5, add 1, and then store the value as 6. Then, the control would return to the first thread that obtained the value 5; the first thread would do the addition and also store the value as 6. Thus, even though the global variable was incremented twice, the result is incorrectly 6 and not 7 because of poor thread synchronization.

Deadlock: A state where threads compete for resources, waiting for each other in a permanent state of stalemate. Deadlock means two (or more) threads cannot move because each one is waiting on a lock the other one is holding. For example, thread A is holding one lock and waiting on another lock held by thread B. At the same time, thread B is waiting on the lock thread A is holding. Thus, neither of the threads will be able to run and the threads are in a state of deadlock. This is a very serious issue because it causes the program to halt.

Livelock results from deadlock. Similar to deadlock except the threads have got trapped into endless thrashing. Some machines have mechanisms to detect deadlock. Often, to resolve deadlock, all the threads drop all their locks and try to move forward. It is possible, though, that after the locks are dropped the threads may run through the same code that produced the initial deadlock and end up in deadlock again. This process of reaching deadlock, dropping locks, and returning to deadlock is called livelock. Even though the threads are not in deadlock, they have stopped accomplishing their tasks.

Starvation: Starvation is when a thread that is not blocked never gets scheduled because of unfair behavior of other threads and the way they communicate.

Busy waiting is a thread problem that does not stop the program from working correctly but affects performance. It occurs when a thread wakes up, realizes it needs to wait on a resource, and continues to check the resource until it is swapped out. This is a problem because nothing useful happens while a thread is continually checking the same thing. The efficiency a program gains by using threads can be lost in busy waiting.

Threads waiting on a resource should sleep until the resource becomes available. This avoids busy waiting.

Over-locking is another problem that affects performance. Consider the following code:

```
wait(semaphore);
/* value is a global variable that needs to have its access synchronized. */
value++;
/* lots of time consuming stuff that does not have to be synchronized */
/* … */
signal(semaphore);
```

While this code will run correctly, it unnecessarily locks resources. It forces time-consuming code to be run serially when it does not need to be. This can create severe performance problems that eliminate the advantages usually gained by using threads.

1.12. What is deadlock? How can I eliminate it?

Deadlock is one of the four particular hazards facing designers of multi-threaded programs. The four hazards are: race, deadlock, livelock, and starvation. All four conditions cannot be eliminated merely by testing. Testing may indicate the presence of a deadlock, but it cannot be relied upon to indicate freedom from deadlock. This must happen by good design.

A clear model that aids good design is CSP or Communicating Sequential Processes (invented by Tony Hoare, who also invented monitors). CSP is a language for describing patterns of interaction. It is supported by an elegant, mathematical theory, a set of proof tools, and an extensive literature. Recent theoretical work has proven that CSP can be used completely reliably in Java. Doug Lea's book describes JCSP, one of the two libraries through which this is achieved. Using JCSP (or CTJ, the other library), it is possible to design Java programs of any complexity that are provably free from deadlock, livelock, race or starvation.

The following solutions must be applied to your particular needs; sometimes it will be impossible to use one or more of them:

- Design your program so the only access to a thread-unsafe object is through a thread-safe object.
- Spawn new threads to handle each part of the transaction. If each thread only locks one object at a time there can be no deadlock.
- Check and back off: If you can test to see if another thread has a lock you want, you can "back off" instead of locking it, allowing it to complete its transaction before trying again. This can't be done with normal Java locks, but you can try to write your own lock object to satisfy them.
- Timeout: If you write your own lock object, it can automatically return from the "lock" method if a certain amount of time elapses.
- Minimize or remove synchronization: If there's no lock, there's no deadlock!

1.13. What is the difference between multithreading and multitasking? What about multiprogramming? Multiprocessing?

Multitasking is running multiple "heavyweight" processes (tasks) by a single OS.

Multithreading is running multiple "lightweight" processes (threads of execution) in a single process / task / program.

Multiprogramming is essentially a synonym for multitasking (though multitasking connotes sharing more resources than just the CPU, and is the more popular term).

Multiprocessing involves using multiple CPUs, either in the same (SMP) or different (MPP) host boxes, to run a program.

Most Java implementations will split threads among different processors if they are running on an SMP box.

1.14. Describe program, process, fork, exec, waitpid.

A program is an executable file residing in a disk file. A program is read into memory and executed by the kernel.

An executing instance of a program is called a process. Every UNIX process is guaranteed to have a unique numeric identifier called the process ID. The process ID is always a nonnegative integer.

There are three primary functions used for process control: fork, exec, and waitpid.

We call fork to create a new process. The new process is a copy of the caller, and we say the caller is the parent and the newly created process is the child. Then fork returns the nonnegative process ID of the new child process to the parent, and it returns 0 to the child. Since fork creates a new process, we say that it is called once (by the parent) but returns twice (in the parent and in the child).

The combination of a fork, followed by an exec, is what some operating systems call spawning a new process. In UNIX the two parts are separated into individual functions.

Using waitpid, we specify which process we want to wait for (the pid argument). The waitpid function also returns the termination status of the child (the status variable). We could examine it to determine exactly how the child terminated.

waitpid(pid, &status, 0)

1.15. What is a fork in UNIX?

The fork() function is used to create a new process from an existing process. The new process is called the child process, and the existing process is called the parent. You can tell which is which by checking the return value from fork(). The parent gets the child's pid (process id) returned to him, but the child gets 0 returned to him.

New process starts execution by returning from fork. Child and parent are nearly identical. Parent gets the child process id from fork. Child gets 0 back from fork. Parent should wait for child to exit.

```
#include <iostream.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (int argc, char **argv)
{
    int i, pid;
    pid = fork();
    cout << pid << endl;
    for ( i = 0; i < 5; i++) {
        cout << getpid() << ' ' << i << endl;
    }
    if ( pid ) wait(NULL);
}
```

1.16. What are the problems with fork? How threads do solves these problems?

In the traditional UNIX model, when a process needs something performed by another entity, it forks a child process and lets the child perform the processing. Most network servers under UNIX, for example, are written this way. Although this paradigm has served well for many years, there are problems with fork:

fork is expensive. Memory is copied from the parent to the child; all descriptors are duplicated in the child, and so on. Current implementations use a technique called copy-on-write, which avoids a copy of the parent's data space to the child until the child needs its own copy; but regardless of this optimization, fork is expensive.

Inter process communication (IPC) is required to pass information between the parent and child after the fork. Information from the parent to the child before the fork is easy, since the child starts with a copy of the parent's data space and with a copy of all the parent's descriptors. But returning information from the child to the parent takes more work.

Threads help with both problems. Threads are sometimes called lightweight processes, since a thread is “lighter weight” than a process. That is, thread creation can be 10-100 times faster than process creation.

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes the problem of synchronization. But more than just the global variables are shared. All threads within a process share:

- process instructions,
- most data,
- open files (for example, descriptors),
- signal handlers and signal dispositions,
- current working directory, and
- user and group IDs.

But each thread has its own:

- thread ID,
- set of registers, including program counter and stack pointer,
- stack (for local variables and return addresses),
- errno,
- signal mask, and
- priority.

1.17. What is a daemon?

It is a background process that listens for requests from other programs. Pronounced DEE-mun or DAY-mun. A process that runs in the background and performs a specified operation at predefined times or in response to certain events. The term daemon is a UNIX term, though many other operating systems provide support for daemons, though they are sometimes called other names. Windows, for example, refers to daemons as System Agents and services.

Typical daemon processes include print spoolers, email handlers, and other programs that perform administrative tasks for the operating system. The term comes from Greek mythology, where daemons were guardian spirits.

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shutdown. They run in the background, because they do not have a controlling terminal. UNIX systems have numerous daemons that perform day-to-day activities. For example, ps, NFS Server daemon.

There are some basic rules to coding a daemon, to prevent unwanted interactions from happening:

- The first thing to do is to call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we are guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.
- Call setsid to create a new session. The three steps occur. The process (1) becomes a session leader of a new session, (2) becomes the process group leader of a new process group, and (3) has no controlling terminal.
- Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted filesystem. Since daemons normally exists until the system is rebooted, if the daemon stays on a mounted filesystem, that filesystem cannot be unmounted. Alternately, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.
- Set the file mode creation mask to 0. The file mode creation mask that is inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
- Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). Exactly which descriptors to close, however, depends on the daemon.

1.18. What is a cron job?

A UNIX command for scheduling jobs to be executed sometime in the future. A cron is normally used to schedule a job that is executed periodically - for example, to send out a notice every morning. It is also a daemon process, meaning that it runs continuously, waiting for specific events to occur.

cron is a UNIX utility that allows tasks to be automatically run in the background at regular intervals by the cron daemon. These tasks are often termed as cron jobs. Crontab (CRON TABle) is a file which contains the schedule of cron entries to be run and at specified times.

You can execute crontab if your name appears in the file /usr/lib/cron/cron.allow. If that file does not exist, you can use crontab if your name does not appear in the file /usr/lib/cron/cron.deny. If only cron.deny exists and is empty, all users can use crontab. If neither file exists, only the root user can use crontab. The allow/deny files consist of one user name per line.

1.19. What is a socket?

A socket is network end point that is the combination of an IP address and a port number. The word socket is used as a metaphor because the information contained in a socket can be plugged directly into the requested process on the target computer on the Internet. A socket can be created to identify an application running anywhere on the Internet.

1.20. What is the main advantage of creating links to a file instead of copies of the file?

The main advantage is not really that it saves disk space (though it does that too) but, rather, that a change of permissions on the file is applied to all the link access points. The link will show permissions of lrwxrwxrwx but that is for the link itself and not the access to the file to which the link points. Thus if you want to change the permissions for a command, such as su, you only have to do it on the original. With copies you have to find all of the copies and change permission on each of the copies.

1.21. Describe hard link.

A great way to give easy access to your files is to create a hard link from your directory. In making a hard link, all you are doing is starting with an existing file and creating a link, which (sort of) places the existing file in another directory. The link does not create a copy of the file; instead, you are creating a second pointer to the same physical file on the disk. Rather than the additional pointer being secondary (like an alias or shortcut in Windows computers), both of the pointers reference the same actual file, so from the perspective of the UNIX system, the file actually resides in two locations.

Because using hard links often requires that you have access to another user's home directory, you may have to use chmod, chgrp, and chown to access another user's directories and files. You can remove hard links just as you remove regular files, by using rm plus the file name. If one user removes the file, the other user can still access the file from his/her directory. Hard links work from file to file within the same file system. To link directories or to link across file systems, you will have to use soft links.

ln

1.22. Describe soft link.

Soft links essentially provide other users with a shortcut to the directory you specify. Like hard links, soft links allow a file to be in more than one place at a time; however, with soft links, there is only one copy of it and, with soft links, you can link directories as well. The linked file or directory is dependent on the original one - that is, if the original file or directory is deleted, the linked file or directory will no longer be available. Soft links are particularly handy because they work for directories as well as individual files, and they work across different file systems, that is, not just within /home, but anywhere on the UNIX system. Like hard links, soft links sometimes require that you have access to another user's directory and files.

ln -s

1.23. What is the difference between a symbolic link and a hard link?

A name/i-node pair is called a link.

A hard link is a link made at inode level that is at a filesystem level. You can only hard link files of the same filesystem.

A symbolic link is just another file whose content is just the path to the "original" file. It can be on another filesystem (and on another computer if it is imported via NFS). If the source file is removed, the link is broken.

A symbolic link is a pointer to a directory entry while a hard link is more like a virtual duplication of a file.

When you attempt to access a symbolic link with a text editor or other program, the kernel redirects the program to the file indicated by the symbolic link's pathname.

Hard link shares the same inode as the file it points to, thus provide different filenames for the same physical file. Hard link can't point to a directory or across different file systems.

1.24. Write a command to find all of the files which have been accessed within the last 30 days.

find / -type f -atime -30 > december.files &

This command will find all the files under root, which is '/', with file type is file. '-atime -30' will give all the files accessed less than 30 days ago. And the output will put into a file call december.files. This command is run in the background due to &.

1.25. What is the most graceful way to get to run level single user mode?

The most graceful way is to use the command init s. If you want to shut everything down before going to single user mode then do init 0 first and from the ok prompt do a boot -s.

1.26. What does the following command line produce? Explain each aspect of this line. $ (date ; ps -ef | awk {print $1}' | sort | uniq | wc -l) >> Activity.log

First let's dissect the line: The date gives the date and time as the first command of the line, this is followed by the a list of all running processes in long form with UIDs listed first, this is the ps -ef. These are fed into the awk which filters out all but the UIDs; these UIDs are piped into sort for no discernible reason and then onto uniq (now we see the reason for the sort - uniq only works on sorted data - if the list is A, B, A, then A, B, A will be the output of uniq, but if it's A, A, B then A, B is the output) which produces only one copy of each UID. These UIDs are fed into wc -l which counts the lines - in this case the number of distinct UIDs running processes on the system. Finally the results of these two commands, the date and the wc -l, are appended to the file "Activity.log". Now to answer the question as to what this command line produces. This writes the date and time into the file Activity.log together with the number of distinct users who have processes running on the system at that time. If the file already exists, then these items are appended to the file, otherwise the file is created.

1.27. In UNIX OS, what is the file server?

The file server is a machine that shares its disk storage and files with other machines on the network.

1.28. What is NFS? What is its job?

NFS stands for Network File System. NFS enables filesystems physically residing on one computer system to be used by other computers in the network, appearing to users on the remote host as just another local disk.

1.29. What is CVS? List some useful CVS commands.

CVS is Concurrent Version System. It is the front end to the RCS (revision control system) which extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories consisting of revision controlled files. These directories and files can be combined together to form a software release.

There are some useful commands that are being used very often. They are:

cvs checkout
cvs update
cvs update -d
cvs add
cvs remove
cvs commit
cvs co -P aegis - to repull the tree

1.30. What do these permissions on a file mean? =-rwxr-xr-x

owner, group, other. Owner has read/write/execute permissions of the file, group and others have read/execute permissions of the file.

1.31. What is a tar file?

tar stands for Tape ARchive. It was originally designed for tape backups, but it is used to create a tar file anywhere on the filesystem. tar creates one "tar file" (also known as a "tarball") out of several files and directories. A tar file is not compressed, it is just a heap of files assembled together in "one container". So, the tar file will take up the same amount of space as all the individual files combined, plus a little extra. A tar file can be compressed by using gzip. Here are some examples:

- % tar -xvf example.tar - Extract the contents of example.tar and display the files as they are extracted.
- % tar -cf backup.tar /home/ftp/pub - Create a tar file named backup.tar from the contents of the directory /home/ftp/pub
- % tar -tvf example.tar - List contents of example.tar to the screen

1.32. Describe gzip.

gzip is the original UNIX ZIP format. It is common to first "tar" files and then compress them afterwards using gzip. These files are normally given the extensions .tar.gz to show that they are tar archives zipped up with gzip. You may also see the extension, .tgz. An archive that is compressed with

gzip is compatible with WinZip and PkZip. So, a file zipped up on a UNIX box can be unzipped with a Windows box. Here are some examples:

To compress a file using gzip, execute the following command:
% gzip filename.tar

(where filename.tar is the name of the file you wish to compress) The result of this operation is a file called filename.tar.gz. By default, gzip will delete the filename.tar file.

To decompress a file using gzip, execute the following command:
% gzip -d filename.tar.gz

The result of this operation is a file called filename.tar. By default, gzip will delete the filename.tar.gz file. You can also decompress the file using the command:
% gunzip filename.tar.gz

This is the same as using the gzip -d command. There are many options that you can use with gzip. Do a man on the utility to learn more.

1.33. How do files differ in Windows, UNIX and Mac?

In Windows applications, a new line is normally stored as a pair of characters: carriage return (CR) and line feed (LF). The character pair bears some resemblance to the typewriter actions of setting a new line. In UNIX applications, a new line is normally stored as a LF character. Macintosh applications use only a CR character to store a new line.

1.34. What is the difference between .so and .a files in UNIX?

.a archive
.o object
.so shared object

1.35. Windows vs. UNIX: Linking dynamic load modules

UNIX and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically

loaded, be aware of how your system works.

In UNIX, a shared object (.so) file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file's code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library (.dll) file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program's memory; instead, the code already uses the DLL's lookup table, and the lookup table is modified at runtime to point to the functions and data.

In UNIX, there is only one type of library file (.a) which contains code from several object files (.o). During the link step to create a .so, the linker may find that it does not know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called .lib). A static library is like a UNIX .a file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the .dll is loaded. So the linker uses the information from the import library to build the lookup table for using identifiers that are not included in the .dll. When an application or a .dll is linked, an import library may be generated, which will need to be used for all future .dll's that depend on the symbols in the application or .dll.

Suppose you are building two dynamic-load modules, B and C, which should share another block of code A. In UNIX, you would *not* pass A.a to the linker for B.so and C.so; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building A.dll will also build A.lib. You *do* pass A.lib to the linker for B and C. A.lib

does not contain code; it just contains information which will be used at runtime to access A's code.

In Windows, using an import library is sort of like using "import spam"; it gives you access to spam's names, but does not create a separate copy. In UNIX, linking with a library is more like "from spam import *"; it does create a separate copy.

A program that is linked to shared libraries will generally run more slowly than a program that is linked to archive libraries. If you use archive libraries, the linker binds into your executable code an actual copy of each library routine you call. If you use shared libraries, the linker merely notes in your executable code that the code calls a routine in a shared library. Then, when the code begins execution, the dynamic loader loads and maps the shared libraries into the process's address space and calls the routines indirectly as they are needed by means of a linkage table. Using shared libraries saves space in the executable file, but at the expense of the time needed to resolve references to the routines in the shared libraries.

The performance impact of shared libraries is likely to be noticeable only if a program makes heavy use of library functions, as many floating-point applications do. If your program seems to be running unacceptably slowly with shared libraries, you may want to find out whether archive libraries make a difference.

The C math library libm and the FORTRAN and Pascal library libcl, however, are provided in both shared and archive versions. The linker by default looks for shared libraries before it looks for archive libraries, so if you want to use the archive library version of libm or libcl, you need to specify the -a archive option to the linker. To do this on the compile command line, specify -Wl,-a, archive.

Shared libraries provide the following advantages:

- dramatically reduced executable size
- reduced link time
- ability to use updated shared libraries without relinking at the expense of a slight increase in program startup time.

Static linking of system libraries (for example, libc.a) into an application is generally not a good idea because the system library code “locks” into the application binary may become incompatible with later releases of the system.

1.36. How would you find all the processes running on your computer?

Running UNIX command ps -ef or ps -aux depending on version.

1.37. List some UNIX commands and their use.

aix df -k /vol1
df -k
UNIX commands
ls
mkdir
chmod 777 *.*
rlogin server -l user Connects to a different machine on your network specified by hostname.
rsh
psexec.exe
uptime
pwd - print working directory
ps -f Shows your currently running processes
ps -ef Shows every currently running processes
sudo The sudo facility allows specified users to run specific commands as root without having to know the root password. sudo requires only the user password to run the command, not the root password.
RCMD
ipconfig The ipconfig command displays system and network information.
ipconfig Displays the IP address
ipconfig/all Displays the hostname, domain name, and IP address.
chmod
chgrp
chown
source xx.cshrc

ps
locate
top
find
prstat
grep send .
gcc -v // to get the version number of the compiler
uname -a // to get the linux kernel number
~ is for home directory
sar system activity reporter
this command is used to collect, report, or save system activity information
du -Sh * Disk usage command

1.38. How do you get a long listing of all files in a directory sorted in reverse order by time?

ls -lta

Look at your system's <errno.h> header and see how it defines errno. Something like the following is typical:

```
#ifdef _REENTRANT
#define errno (*_errno())
#else
extern int errno;
#endif
```

1.39. What are shared libraries?

Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwards automatically use the new shared library. It is actually much more flexible and sophisticated than this, because the approach used by Linux permits you to:

- update libraries and still support programs that want to use older, non-backward-compatible versions of those libraries;
- override specific libraries or even specific functions in a library when executing a particular program.

- do all this while programs are running using existing libraries.

1.40. What is LD_LIBRARY_PATH used for?

LD_LIBRARY_PATH is an environment variable you set to give the run-time shared library loader (ld.so) an extra set of directories to look for when searching for shared libraries. Multiple directories can be listed, separated with a colon (:). This list is prepended to the existing list of compiled-in loader paths for a given executable, and any system default loader paths. For security reasons, LD_LIBRARY_PATH is ignored at runtime for executables that have their setuid or setgid bit set. This severely limits the usefulness of LD_LIBRARY_PATH.

1.41. Why was LD_LIBRARY_PATH invented?

There were a couple good reasons why it was invented:

- To test out new library routines against an already compiled binary (for either backward compatibility or for new feature testing).
- To have a short term way out in case you wanted to move a set of shared libraries to another location.

As an often unwanted side effect, LD_LIBRARY_PATH will also be searched at link (ld) stage after directories specified with -L (also if no -L flag is given). Some good examples of how LD_LIBRARY_PATH is used:

When upgrading shared libraries, you can test out a library before replacing it. In a similar vein, in case your upgrade program depends on shared libraries and may freak out if you replace a shared library out from under it, you can use LD_LIBRARY_PATH to point to a directory with copy of a shared libraries and then you can replace the system copy without worry. You can even undo things should things fail by moving the copy back.

X11 uses LD_LIBRARY_PATH during its build process. X11 distributes its fonts in “bdf” format, and during the build process it needs to “compile” the bdf files into “pcf” files. LD_LIBRARY_PATH is used to point to the build lib directory so it can run bdftopcf during the build stage before the shared libraries are installed.

Perl can be installed with most of its core code as a shared library. This is handy if you embed Perl in other programs - you can compile them so they use the shared library and so you'll save memory at run time. However Perl uses Perl scripts at various points in the build and install process. The 'perl' binary won't run until its shared libraries are installed, unless LD_LIBRARY_PATH is used to bootstrap the process.

1.42. What command can be used to find out idle processes?

ps

1.43. Which command can be used to read from logged data?

sar

1.44. What are the different types of files in UNIX?

There are several kinds of UNIX files: regular files, directories, symbolic links, special files, named pipes (FIFOs), and sockets.

1.45. What are regular files?

Regular files contain bytes of data, organized into a linear array. Any byte or sequence of bytes may be read or written. Reads and writes start at a byte location specified by the file offset, which can be set to any value (even beyond the end of the file). Regular files are stored on disk. It is not possible to insert bytes into the middle of a file (spreading the file apart), or to delete bytes from the middle (closing the gap). As bytes are written onto the end of a file, it gets bigger, one byte at a time. A file can be shrunk or enlarged to any length, discarding bytes or adding bytes of zeroes. Two or more processes can read and write the same file concurrently. The results depend on the order in which the individual I/O requests occur and are in general unpredictable. To maintain order, there are file-locking facilities and semaphores, which are system-wide flags that processes can test and set.

1.46. What is an i-node?

Files do not have names; they have numbers called i-numbers. An i-number is an index into an array of i-nodes, kept at the front of each region of disk

that contains a UNIX file system. Each i-node contains important information about one file. Interestingly, this information does not include either the name or the data bytes. It does include the following: type of file (regular, directory, socket, etc.); number of links; owner's user and group ID; three sets of access permissions - for the owner, the group, and others; size in bytes; time of last access, last modification, and status change (when the i-node itself was last modified); and, of course, pointers to disk blocks containing the file's contents.

1.47. Describe directories and symbolic links.

Since it is inconvenient to refer to files by i-number, directories are provided to allow names to be used. In practice, a directory is almost always used to access a file.

Each directory consists, conceptually, of a two-column table, with a name in one column and its corresponding i-number in the other column. A name/i-node pair is called a link. When the UNIX kernel is told to access a file by name, it automatically looks in a directory to find the i-number. Then it gets the corresponding i-node, which contains more information about the file (such as who can access it). If the data itself is to be accessed, the i-node tells where to find it on the disk.

Directories, which are almost like regular files, occupy an i-node and have data. Therefore, the i-node corresponding to a particular name in a directory could be the i-node of another directory. This allows users to arrange their files into the hierarchical structure that's familiar to users of UNIX. A path such as memo/july/smith instructs the kernel to get the i-node of the current directory to locate its data bytes, find memo among those data bytes, take the corresponding i-number, get that i-node to locate the memo directory's data bytes, find july among those, take the corresponding i-number, get the i-node to locate the july directory's data bytes, find smith, and, finally, take the corresponding i-node, the one associated with memo/july/smith.

In following a relative path (one that starts with the current directory), how does the kernel know where to start? It simply keeps track of the i-number of the current directory for each process. When a process changes its current directory, it must supply a path to the new directory. That path leads

to an i-number, which then is saved as the i-number of the new current directory.

An absolute path begins with a / and starts with the root directory. The kernel simply reserves an i-number (2, say) for the root directory. This is established when a file system is first constructed. There is a system call to change a process's root directory (to an i-number other than 2).

Because the two-column structure of directories is used directly by the kernel (a rare case of the kernel caring about the contents of files), and because an invalid directory could easily destroy an entire UNIX system, a program (even if run by the superuser) cannot write a directory as if it were a regular file. Instead, a program manipulates a directory by using a special set of system calls. After all, the only legal writing actions are to add or remove a link.

It is possible for two or more links, in the same or different directories, to refer to the same i-number. This means that the same file may have more than one name. There is no ambiguity when accessing a file by a given path, since only one i-number will be found. It might have been found via another path also, but that's irrelevant. When a link is removed from a directory, however, it isn't immediately clear whether the i-node and the associated data bytes can be thrown away too. That is why the i-node contains a link count. Removing a link to an i-node merely decrements the link count; when the count reaches zero, the kernel discards the file.

There is no structural reason why there can't be multiple links to directories as well as to regular files. However, this complicates the programming of commands that scan the entire file system, so most kernels outlaw it.

Multiple links to a file using i-numbers work only if the links are in the same file system, as i-numbers are unique only within a file system. To get around this, there are also symbolic links, which put the path of the file to be linked to in the data part of an actual file. This is more overhead than just making a second directory link somewhere, but it's more general. You don't read and write these symbolic-link files, but instead use special system calls just for symbolic links.

1.48. What are special files?

A special file is typically some type of device (such as a CD-ROM drive or communications link). Sometimes named pipes are considered special files.

There are two principal kinds of device special files: block and character. Block special files follow a particular model: The device contains an array of fixed-size blocks (say, 4096 bytes each), and a pool of kernel buffers are used as a cache to speed up I/O. Character special files don't have to follow any rules at all. They might do I/O in very small chunks (characters) or very big chunks (disk tracks), and so they are too irregular to use the buffer cache.

The same physical device could have both block and character special files, and, in fact, this is usually true for disks. Regular files and directories are accessed by the file-system code in the kernel via a block special file, to gain the benefits of the buffer cache. Sometimes, primarily in high-performance applications, more direct access is needed. For instance, a database manager can bypass the file system entirely and use a character special file to access the disk (but not the same area that's being used by the file system). Most UNIX systems have a character special file for this purpose that can directly transfer data between a process's address space and the disk using direct memory access (DMA), which can result in orders-of-magnitude better performance. More robust error detection is another benefit, since the indirectness of the buffer cache tends to make error detection difficult to implement.

A special file has an i-node, but there aren't any data bytes on disk for the i-node to point to. Instead, that part of the i-node contains a device number. This is an index into a table used by the kernel to find a collection of subroutines called a device driver.

When a system call is executed to perform an operation on a special file, the appropriate device driver subroutine is invoked. What happens then is entirely up to the designer of the device driver; since the driver runs in the kernel, and not as a user process, it can access - and perhaps modify - any part of the kernel, any user process, and any registers or memory of the

computer itself. It is relatively easy to add new device drivers to the kernel, so this provides a hook with which to do many things besides merely interfacing to new kinds of I/O devices. It's the most popular way to get UNIX to do something its designers never intended it to do. Think of it as the approved way to do something wild.

1.49. What are signals?

The kernel can send a signal to a process. A signal can be originated by the kernel itself, sent from a process to itself, sent from another process, or sent on behalf of the user.

Signals are a technique used to notify a process that some conditions have occurred. For example, if a process divides by zero, the signal whose name is SIGFPE (floating point error) is sent to the process. The process has three choices for dealing with the signal:

1. Ignore the signal. This is not recommended for signals that denote a hardware exception, such as dividing by zero, or referencing memory outside the address space of the process, since the results are undefined.
2. Let the default action occur. For a divide by zero, the default is to terminate the process.
3. Provide a function that is called when the signal occurs. By providing a function of our own, we will know when the signal occurs and we can handle it as we wish.

An example of a kernel-originated signal is a segmentation-violation signal, sent when a process attempts to access memory outside of its address space.

An example of a signal sent by a process to itself is an abort signal, sent by the abort function to terminate a process with a core dump.

An example of a signal sent from one process to another is a termination signal, sent when one of several related processes decides to terminate the whole family.

Finally, an example of a user-originated signal is an interrupt signal, sent to all processes created by the user when he or she types Ctrl-c.

There are about 28 types of signals (some versions of UNIX have a few more or a few less). For all but 2, the kill and stop signals, a process can control what happens when it receives the signal. It can accept the default action, which usually results in termination of the process; it can ignore the signal; or it can catch the signal and execute a function, called a signal handler, when the signal arrives. The signal type (SIGALRM, say) is passed as an argument to the handler. There isn't any direct way for the handler to determine who sent the signal, however. When the signal handler returns, the process resumes executing at the point of interruption. Two signals are left unused by the kernel. These may be used by an application for its own purposes.

1.50. Describe process-IDs, process groups, and sessions.

Every process has a process-ID, which is a positive integer. At any instant, these are guaranteed to be unique. Every process but one has a parent.

A process's system data also records its parent-process-ID, the process-ID of its parent. If a process is orphaned because its parent terminated before it did, its parent-process-ID is changed to 1 (or some other fixed number). This is the process-ID of the initialization process (init), created at boot time, which is the ancestor of all other processes. In other words, the initialization process adopts all orphans.

Sometimes programmers choose to implement a subsystem as a group of related processes instead of as a single process. The UNIX kernel allows these related processes to be organized into a process group. Process groups are further organized into sessions.

One of the session members is the session leader, and one member of each process group is the process-group leader. For each process, UNIX keeps track of the process IDs of its process-group leader and session leader so that a process can locate itself in the hierarchy.

The kernel provides a system call to send a signal to each member of a process group. Typically, this would be used to terminate the entire group, but any signal can be broadcast in this way.

UNIX shells generally create one session per login. They usually form a separate process group for the processes in a pipeline; in this context a process group is also called a job. But that's just the way shells like to do things; the kernel allows more flexibility so that any number of sessions can be formed within a single login, and each session can have its process groups formed any way it likes, as long as the hierarchy is maintained.

It works this way: Any process can resign from its session and then become a leader of its own session (of one process group containing only one process). It can then create child processes to round out the new process group. Additional process groups within the session can be formed and processes assigned to the various groups. The assignments can be changed, too. Hence, a single user could be running, say, a session of 10 processes formed into, say, three process groups.

A session, and thus the processes in it, can have a controlling terminal, which is the first terminal device opened by the session leader; the session leader then becomes the controlling process. Normally, the controlling terminal for a user's processes is the terminal from which the user logged in. When a new session is formed, the processes in the new session no longer have a controlling terminal, until one is opened. Not having a controlling terminal is typical of so-called daemons (Web servers, the cron facility, etc.) which, once started, run divorced from whatever terminal was used to start them. "Demon" and "daemon" are two spellings of the same word, but the former connotes an evil spirit, whereas the latter connotes a supernatural being somewhere between God and man. Misbehaving daemons may properly be referred to as demons.

It is possible to organize things so that only one process group in a session - the foreground job - has access to the terminal and to move process groups back and forth between foreground and background. This is called job control. A background process group that attempts to access the terminal is suspended until moved to the foreground. This prevents it from stealing input from the foreground job or scribbling on the foreground process group's output.

If job control isn't being used (generally, only shells use it), all the processes in a session are effectively foreground processes and are equally free to access the terminal, even if havoc ensues.

The terminal device driver sends interrupt, quit, and hangup signals coming from a terminal to every process in the foreground process group for which that terminal is the controlling terminal. For example, unless precautions are taken, hanging up a terminal (e.g., closing a telnet connection) will terminate all of the user's processes in that group. To prevent this, a process can arrange to ignore hang-ups.

Additionally, when a session leader (controlling process) terminates for any reason, all processes in the foreground process group are sent a hangup signal. So simply logging out usually terminates all of a user's processes unless specific steps have been taken to either make them background processes or otherwise make them immune to the hangup signal.

In summary, there are four process-IDs associated with each process:

- process-ID: Positive integer that uniquely identifies this process.
- parent-process-ID: Process-ID of this process's parent.
- process-group-ID: Process-ID of the process-group leader. If equal to the process-ID, this process is the group leader.
- session-leader-ID: Process-ID of the session leader.

1.51. Permissions

A user-ID is a positive integer that is associated with a user's login name in the password. file (/etc/passwd). When a user logs in, the login command makes this ID the user-ID of the first process created, the login shell. Processes descended from the shell inherit this user-ID.

Users are also organized into groups (not to be confused with process groups), which have IDs too, called group-IDs. A user's login group-ID is made the group-ID of his or her login shell.

Groups are defined in the group file (/etc/group). While logged in, a user can change to another group of which he or she is a member. This changes the group-ID of the process that handles the request (normally the shell, via

the newgrp command), which then is inherited by all descendent processes. As a user may be a member of more than one group, a process also has a list of supplementary group-IDs. For most purposes these are also checked when a process's group permissions are at issue, so a user doesn't have to be constantly switching groups manually.

These two user and group login IDs are called the real user-ID and the real group-ID because they are representative of the real user, the person who is logged in. Two other IDs are also associated with each process: the effective user-ID and the effective group-ID. These IDs are normally the same as the corresponding real IDs, but they can be different, as we shall see shortly.

The effective ID is always used to determine permissions. The real ID is used for accounting and user-to-user communication. One indicates the user's permissions; the other indicates the user's identity.

Each file (regular, directory, socket, etc.) has, in its i-node, an owner user-ID (owner for short) and an owner group-ID (group for short). The i-node also contains three sets of three permission bits (nine bits in all). Each set has one bit for read permission, one bit for write permission, and one bit for execute permission. A bit is 1 if the permission is granted and 0 if not. There is a set for the owner, for the group, and for others (not in either of the first two categories). The following table shows the bit assignments (bit 0 is the rightmost bit).

Permission Bits

Bit	Meaning
8	owner read
7	owner write
6	owner execute
5	group read
4	group write
3	group execute
2	others read
1	others write
0	others execute

Permission bits are frequently specified using an octal (base 8) number. For example, octal 775 would mean read, write, and execute permission for the owner and the group, and only read and execute permission for others. The ls command would show this combination of permissions as rwxrwxr-x; in binary it would be 111111101, which translates directly to octal 775. And in decimal it would be 509, which is useless, because the numbers don't line up with the permission bits.

The permission system determines whether a given process can perform a desired action (read, write, or execute) on a given file. For regular files, the meaning of the actions is obvious. For directories, the meaning of read is obvious. "Write" permission on a directory means the ability to issue a system call that would modify the directory (add or remove a link). "Execute" permission means the ability to use the directory in a path (sometimes called "search" permission). For special files, read and write permissions mean the ability to execute the read and write system calls. What, if anything, that implies is up to the designer of the device driver. Execute permission on a special file is meaningless.

The permission system determines whether permission will be granted using this algorithm:

- If the effective user-ID is zero, permission is instantly granted (the effective user is the superuser).
- If the process's effective user-ID matches the file's user-ID, then the owner set of bits is used to see if the action will be allowed.
- If the process's effective group-ID or one of the supplementary group-IDs matches the file's group-ID, then the group set of bits is used.
- If neither the user-IDs nor group-IDs match, then the process is an "other" and the third set of bits is used.

The steps go in order, so if in step 3 access is denied because, say, write permission is denied for the group, then the process cannot write, even though the "other" permission (step 4) might allow writing. It might be unusual for the group to be more restricted than others, but that's the way it works. (Imagine an invitation to a surprise party for a team of employees.

Everyone except the team should be able to read it.)

There are other actions, which might be called “change i-node,” that only the owner or the superuser can do. These include changing the user-ID or group-ID of a file, changing a file’s permissions, and changing a file’s access or modification times. As a special case, write permission on a file allows setting of its access and modification times to the current time.

Occasionally, we want a user to temporarily take on the privileges of another user. For example, when we execute the passwd command to change our password, we would like the effective user-ID to be that of the superuser, because only root can write into the password file. This is done by making root (the superuser's login name) the owner of the passwd command (i.e., the regular file containing the passwd program) and then turning on another permission bit in the passwd command's i-node, called the set-user-ID bit.

Executing a program with this bit on changes the effective user-ID of the process to the owner of the file containing the program. Since it's the effective user-ID, rather than the real user-ID, that determines permissions, this allows a user to temporarily take on the permissions of someone else. The set-group-ID bit is used in a similar way.

Since both user-IDs (real and effective) are inherited from parent process to child process, it is possible to use the set-user-ID feature to run with an effective user-ID for a very long time. That's what the su command does.

There is a potential loophole. Suppose you do the following: Copy the sh command to your own directory (you will be the owner of the copy). Then use chmod to turn on the set-user-ID bit and chown to change the file's owner to root. Now execute your copy of sh and take on the privileges of root! Fortunately, this loop-hole was closed a long time ago. If you're not the superuser, changing a file's owner automatically clears the set-user-ID and set-group-ID bits.

Other Process Attributes

A few other interesting attributes are recorded in a process's system data segment.

There is one open file descriptor (an integer from 0 through 1000 or so) for each file (regular, special, socket, or named pipe) that the process has opened, and two for each unnamed pipe that the process has created. A child doesn't inherit open file descriptors from its parent, but rather duplicates of them. Nonetheless, they are indices into the same system-wide

open file table, which among other things means that parent and child share the same file offset (the byte where the next read or write takes place).

A process's priority is used by the kernel scheduler. Any process can lower its priority via the system call nice; a superuser process can raise its priority via the same system call. Technically speaking, nice sets an attribute called the nice value, which is only one factor in computing the actual priority.

A process's file size limit can be (and usually is) less than the system-wide limit; this is to prevent confused or uncivilized users from writing runaway files. A superuser process can raise its limit.

1.52. Interprocess Communication

In the oldest (pre-1980) UNIX systems, processes could communicate with one another via shared file offsets, signals, process tracing, files, and pipes. Then named pipes (FIFOs) were added, followed by semaphores, file locks, messages, shared memory, and networking sockets. None of these eleven mechanisms is entirely satisfactory. That's why there are eleven! There are even more because there are two versions each of semaphores, messages, and shared memory. There's one very old collection from AT&T's System V UNIX, called "System V IPC," and one much newer, from a real-time standards group that convened in the early 1990s, called "POSIX IPC." Almost every version of UNIX (including FreeBSD and Linux) has the oldest eleven mechanisms, and the main commercial versions of UNIX (e.g., Sun's Solaris or HP's HP/UX) have all fourteen.

Shared file offsets are rarely used for interprocess communication. In theory, one process could position the file offset to some fictitious location in a file, and a second process could then find out where it points. The location (a number between, say, 0 and 100) would be the communicated data. Since the processes must be related to share a file offset, they might as well just use pipes.

Signals are sometimes used when a process just needs to poke another. For example, a print spooler could signal the actual printing process whenever a print file is spooled. But signals don't pass enough information to be helpful in most applications. Also, a signal interrupts the receiving process, making

the programming more complex than if the receiver could get the communication when it was ready. Signals are mainly used just to terminate processes or to indicate very unusual events.

With process tracing, a parent can control the execution of its child. Since the parent can read and write the child's data, the two can communicate freely. Process tracing is used only by debuggers, since it is far too complicated and unsafe for general use.

Files are the most common form of interprocess communication. For example, one might write a file with a process running vi and then execute it with a process running python. However, files are inconvenient if the two processes are running concurrently, for two reasons: First, the reader might outrace the writer, see an end-of-file, and think that the communication is over. (This can be handled through some tricky programming.) Second, the longer the two processes communicate, the bigger the file gets. Sometimes processes communicate for days or weeks, passing billions of bytes of data. This would quickly exhaust the file system.

Using an empty file for a semaphore is also a traditional UNIX technique. This takes advantage of some peculiarities in the way UNIX creates files.

Finally, something that is usually recommended: Pipes solve the synchronization problems of files. A pipe is not a type of regular file; although it has an i-node, there are no links to it. Reading and writing a pipe is somewhat like reading and writing a file, but with some significant differences: If the reader gets ahead of the writer, the reader blocks (stops running for a while) until there is more data. If the writer gets too far ahead of the reader, it blocks until the reader has a chance to catch up, so the kernel doesn't have too much data queued. Finally, once a byte is read, it is gone forever, so long-running processes connected via pipes don't fill up the file system.

Pipes are well known to shell users, who can enter command lines like

```
ls | wc
```

to see how many files they have. The kernel facility, however, is far more general than what the shell provides.

Pipes, however, have three major disadvantages:

- First, the processes communicating over a pipe must be related, typically parent and child or two siblings. This is too constraining for many applications, such as when one process is a database manager and the other is an application that needs to access the database.
- The second disadvantage is that writes of more than a locally set maximum (4096 bytes, say) are not guaranteed to be atomic, prohibiting the use of pipes when there are multiple writers - their data might get intermingled.
- The third disadvantage is that pipes might be too slow. The data has to be copied from the writing user process to the kernel and back again to the reader. No actual disk I/O is performed, but the copying alone can take too long for some critical applications. It is because of these disadvantages that fancier schemes have evolved.

Named pipes, also called FIFOs, were added to solve the first disadvantage of pipes. FIFO stands for “first-in-first-out.” A named pipe exists as a special file, and any process with permission can open it for reading or writing. Named pipes are easy to program with. What’s wrong with named pipes? They don’t eliminate the second and third disadvantage of pipes: Interleaving can occur for big writes, and they are sometimes too slow. For the most critical applications, the newer interprocess communication features (e.g., messages or shared memory) can be used, but not nearly as easily.

A named pipe is a special file that is used to transfer data between unrelated processes. One or more processes write to it, while another process reads from it. Named pipes are visible in the file system and may be viewed with ‘ls’ like any other file. Named pipes may be used to pass data between unrelated processes, while normal (unnamed) pipes can only connect parent/child processes (unless you try very hard). Named pipes are strictly unidirectional, even on systems where anonymous pipes are bidirectional (full-duplex).

A semaphore is a counter that prevents two or more processes from accessing the same resource at the same time. Files can be used for semaphores too, but the overhead is far too great for many applications. UNIX has two completely different semaphore mechanisms, one part of System V IPC, and the other part of POSIX IPC.

A file lock, in effect a special-purpose semaphore, prevents two or more processes from accessing the same segment of a file. It's usually only effective if the processes bother to check; that weak form is called advisory locking. The stronger form, mandatory locking, is effective whether a process checks or not, but it's nonstandard and available only in some UNIX systems.

A message is a small amount of data (500 bytes, say) that can be sent to a message queue. Messages can be of different types. Any process with appropriate permissions can receive messages from a queue. It has lots of choices: either the first message, or the first message of a given type, or the first message of a group of types. As with semaphores, there are System V IPC messaging system calls, and completely different POSIX IPC system calls.

Shared memory potentially provides the fastest interprocess communication of all. The same memory is mapped into the address spaces of two or more processes. As soon as data is written to the shared memory, it is instantly available to the readers. A semaphore or a message is used to synchronize the reader and writer. Sure enough, there are two versions of shared memory.

The best of all is: networking interprocess communication, using a group of system calls called sockets. (Other system calls in this group have names like bind, connect, and accept.) Unlike all of the other mechanisms we have talked about, the process you are communicating with via a socket does not have to be on the same machine. It can be on another machine, or anywhere on a local network or the Internet. That other machine does not even have to be running UNIX. It can be running Windows or whatever. It could be a networked printer or radio, for that matter.

1.53. What are different versions of UNIX ?

Ken Thompson and Dennis Ritchie began UNIX as a research project at AT&T Bell Laboratories in 1969, and shortly thereafter it started to be widely used within AT&T for internal systems (e.g., to automate telephone-repair call centers). AT&T wasn't then in the business of producing commercial computers or of setting commercial software, but by the early 1970s it did make UNIX available to universities for educational purposes, provided the source code was disclosed only to other universities who had their own license. By the late 1970s AT&T was licensing source code to commercial vendors, too.

Many, if not most, of the licensees made their own changes to UNIX, to port it to new hardware, to add device drivers, or just to tune it. Perhaps the two most significant changes were made at the University of California at Berkeley: networking system-calls ("sockets") and support for virtual memory. As a result, universities and research labs throughout the world used the Berkeley system (called BSD, for Berkeley Software Distribution), even though they still got their licenses from AT&T. Some commercial vendors, notably Sun Microsystems, started with BSD UNIX as well. Bill Joy, a founder of Sun, was a principal BSD developer.

1.54. How do you list all the files and sub-directories in the file system?

ls -lR
List all the files and their sizes.

1.55. Write a command to list all of the files in the current directory tree that have a name ending in .c.

find . -name "*.c" -print

1.56. Describe the C compilation process.

The C compiler, cc, translates C source programs into object modules or into executable modules. An executable module is ready to be loaded and executed. The compilation process proceeds in stages. In the first stage, a preprocessor expands macros and includes header files. The compiler then

makes several passes through the code to translate the code first to the assembly language of the target machine and then into machine code. The result is an object module consisting of machine code and tables of unresolved references. The final stage of compilation links a collection of object modules together to form the executable module with all references resolved. The executable contains exactly one main function.

The following command compiles mine.c and produces an executable mine.

cc -o mine mine.c

If the -o mine option is omitted, the C compiler produces an executable called a.out. Use the -o option to avoid the noninformative default name.

The following mine.c source file contains an undefined reference to the serr function.

```
void serr(char *msg);
void main(int argc, char *argv[ ])
{   serr("This program does not do much.\n");
}
```

When mine.c is compiled, the C compiler displays a message indicating that serr is an unresolved reference and does not produce an executable.

Most programs are not contained in a single source file, requiring that the sources from multiple files be linked together. All of the source files to be compiled may be specified in a single cc command. Alternatively, the user can compile the source into separate object modules and link these object modules to form an executable modules in a separate step.

Suppose that the serr function is contained in the source file minelib.c. The following command compiles the mine.c source file with minelib.c to produce an executable module called mine.

cc -o mine mine.c minelib.c

The -c option of cc causes the C compiler to produce an object module rather than an executable. An object module cannot be loaded into memory or executed until it is linked to libraries and other modules to resolve references. A misspelled variable or missing library function may not be detected until that object is linked into an executable.

The following command produces the object module mine.o.

```
cc -c mine.c
```

When the -c option is used, the C compiler produces an object module named with a .o extension. The mine.o produced by the cc command can later be linked with another object file (e.g., minelib.o) to produce an executable.

The following links the object modules mine.o and minelib.o to produce the executable mine.

```
cc -o mine mine.o minelib.o
```

1.57. Describe the make process.

The make utility allows users to recompile incrementally a collection of program modules. Even for a simple compile, make is convenient and helps avoid mistakes.

In order to use make, the user must specify dependencies among modules in a description file. The default description filenames are makefile and Makefile. When the user types make, the make utility looks for makefile or Makefile in the current directory and checks this description file to see if anything needs updating.

The description file describes the dependency relationships that exist between various program modules. Lines starting with # are comments. The dependencies in the description file have the form:

```
target: components
```

TAB rule

The first line is called a dependency and the second line is called a rule. The first character on a rule line in a description file must be the TAB character. A dependency may be followed by one or more rule lines.

In cc -o mine mine.o minelib.o executable mine depends on the object files mine.o and minelib.o. The following segment describes that dependency relationship.

```
mine: mine.o minelib.o
    cc -o mine mine.o minelib.o
```

The description says that mine depends on mine.o and minelib.o. If either of these latter two files has been modified since mine was last changed, mine should be updated by executing cc -o mine mine.o minelib.o. If the description is contained in a file called makefile, just type the word make to perform the needed updates. The description lines must start with a TAB, so there is an invisible TAB character before the cc.

The filename preceding the colon, mine, is called a target. The target depends on its components (e.g., mine.o and minelib.o). The line following the dependency specification is the command or rule for updating the target if it is older than any of its components. Thus, if mine.o or minelib.o change, execute make to update mine.

In more complicated situations, a given target can depend on components that are themselves targets. The following makefile description file has three targets.

```
my:     my.o mylib.o
        cc -o my my.o mylib.o
my.o:   my.c myinc.h
        cc -c my.c
mylib.o: mylib.c myinc.h
        cc -c mylib.c
```

Just type make to do the required updates.

A description file can also contain macro definitions of the form:

NAME = value

Whenever $(NAME) appears in the description file, value is substituted before the phrase is processed. Do not use tabs in macros.

1.58. Write a typical makfile.

```
# Compiler/Linker
CC = gcc
LD = $(CC)
# compiler/linker flags
CFLAGS = -g -I.. -I../../../shared/base
LDFLAGS = -g
# files removal
RM = /bin/rm -f
# library to use when linking the main program
LIBS = -L../../../dist/lib -lbase -lprotocol -lalert \
      -lrules -lcclient-glue -lutils-glue -L/usr/local/pgsql/lib \
      ../../../dist/lib/expat.a \
      ../../../dist/lib/utils.a ../../../dist/lib/c-client.a \
      -lpam -lcrypto -lssl -lpcap -lpq -lpthread
# program's object files
PROG_OBJS = testAlertAPI.o
# program's executable
PROG = alert
# top-level rule
all: $(PROG)
$(PROG): $(PROG_OBJS)
$(LD) $(LDFLAGS) $(PROG_OBJS) $(LIBS) -o $(PROG)
# compile C source files into object files.
%.o: %.cpp
$(CC) $(CFLAGS) -c $<
# clean everything
```

```
clean:
$(RM) $(PROG_OBJS) $(PROG)
```

Write a typical makfile.

```
DEPTH = ../..
LIBRARY_NAME = alert
INC = -I$(DEPTH)/shared/base \
    -I$(DEPTH)/thirdparty/postgresql-8.0.1/include \
    -I/usr/local/pgsql/include
LIBS = -lbase -L/usr/local/pgsql/lib \
    -L$(DEPTH)/thirdparty/postgresql-8.0.1/lib -lpq
include $(DEPTH)/config/rules.mk
```

1.59. Write a UNIX command to list the biggest file in any directory.

This will list the biggest file on a filesystem or /directory

```
ls -lsR /filesystem | sort -rn
```

Here we have taken a list of our files sorted alphabetically, each with the size in blocks. We have run this to the standard input of the sort command asking it to sort numerically in reverse order (largest first). This output has then been run into the command head which gives us the first few lines. In this case we have asked head for the first 5 lines. Thus this command gives us the names and sizes of our 5 largest files.

```
ls -lsR | sort -rn | head -5
```

```
find /"filesystem name"  -name "*" -type f -exec ls -lR {} \; | awk '{print $5" " $9}'|sort -nr
```

1.60. What are the advantages of logging in to UNIX system?

Logging in serves a few purposes, including giving you access to your files and configurations. It also, however, keeps you from inadvertently accessing someone else's files and settings, and it keeps you from making changes to the system itself. After you have logged in, you will be located

in your home directory, which is where your personal files and settings are stored. Virtually all UNIX systems require passwords to help ensure that your files and data remain your own and that the system itself is secure from vandals and hackers.

1.61. What are the guidelines for creating a password?

In addition to following any password guidelines your system administrator mandates, you should choose a password that is:

At least six characters long
Easy for you to remember
Not a word or name in any dictionary in any language
A combination of capital and lowercase letters, numbers, and symbols
Not similar to your userid
Not identical or similar to one you have used recently
Not your telephone number, birth date, kid's birth date, anniversary, mother's maiden name, or anything else that anyone might associate with you.

1.62. What are the guidelines for changing a password?

You should normally change your password lots of times as follows:

- You will probably want to change the password provided by your system administrator after you log in for the first time.
- You will probably change your password at regular intervals, as many UNIX systems require that you change your password every so often - every 30 or 60 days or so is common.
- You might also change your password voluntarily if you think that someone might have learned it or if you tell anyone your password (although you really should not do that anyway).

1.63. Can you really break up the UNIX system?

In general, no. When you log into a UNIX system and use your personal userid, the worst you can do is remove your own directories and files. As long as you are logged in as yourself, commands you type would not affect anything critical to the UNIX system, only your personal directories and

files. Score one for UNIX - as an average user, you cannot break the system. With Windows or Macintosh, though, it is a different story.

If you have system administrator rights, meaning that you can log in as root (giving you access to all the system directories and files), you can do a lot of damage if you are not extremely careful. For this reason, do not log in as root unless you absolutely have to.

1.64. Where do you use su command?

Occasionally, you may need to log in with a userid other than your own or need to re-log in with your own userid. For example, you might want to check configuration settings that you have changed before logging out to make sure that they work. Or, if you change your shell, you might want to check it before you log out.

You can use the su (substitute user) command to either log in as another user or to start a new login shell.

su newuserid /* to log in as a different user with su
su - yourid /* to start a new login shell with su. The addition of the hyphen (-) will force a new login shell and set all of the environment variables and defaults according to the settings for the user.

If you have root access and telnet to the system to administer it, you should use su to provide a little extra security. Rather than logging in directly as root and leaving the remote possibility of having your password stolen (or sniffed) over the network, log in as yourself, then use su (with no other information) to change to root.

sed
sed lets you make multiple changes to files without ever opening an editor.
awk
While sed is line-oriented, awk is field-oriented and is ideal for manipulating database or comma-delimited files. For example, if you have an address book file, you can use awk to find and change information in fields you specify.

1.65. What is piping?

The process, in which you connect the output of one program to the input of another, is called piping. Depending on what you want to do, you can pipe together as many commands as you want - with the output of each acting as the input of the next.

1.66. What are the common UNIX directories and their contents?

/bin Essential programs and commands for use by all users
/etc System configuration files and global settings
/home Home directories for users
/sbin Programs and commands needed for system boot
/tmp Temporary files
/usr/bin Commands and programs that are less central to basic UNIX system functionality than those in /bin but were installed with the system
/usr/local Most files and data that were developed or customized on the system
/usr/local/bin Locally developed or installed programs
/usr/local/man Manual (help) pages for local programs
/usr/share/man Manual (help) pages
/var Changeable data, including system logs, temporary data from programs, and user mail storage

1.67. Describe common UNIX shells.

When you access UNIX, the first thing you see is the prompt, called the shell prompt, which is where you interact with the UNIX system. The shell determines how easily you can enter and reenter commands and how you can control your environment. You are not stuck with one shell - that is, on most systems you can choose to use different shells that have different features and capabilities.

sh - This shell, which is the original UNIX shell (often called the Bourne shell), is great for scripting but lacks a lot of the flexibility and power for interactive use that you might want. For example, it does not have features like command completion, email checking, or aliasing.

csh and tcsh - This family of shells adds great interactive uses but discards

the popular scripting support that sh offers in favor of C programming-ish syntax. Because of the C syntax, this shell is often just called C shell.

ksh, bash, and zsah - These provide a good blend of scripting and interactive capabilities, but they stem from different sources (bash is most similar to sh, hence the Bourne Again SHell name).

1.68. What is the difference between cmp and diff?

diff tells you specifically where two files differ, not just that they differ and at which point the differences start. sdiff presents the two files onscreen so that you can visually compare them.

1.69. What are aliases in UNIX?

Aliases are nicknames of sorts that you use to enter commands more easily. For example, if you frequently use the command

mail -s “Lunch today?” Deb < .signature

you could set an alias for this command and call it lunch. Then, in the future, all you have to do is type in lunch, and the result is the same as if you typed in the longer command.

1.70. What are environment variables?

Environment variables are settings in the UNIX system that specifies how you, your shell, and the UNIX system interact. When you log into the UNIX system, it sets up your standard environment variables

1.71. What is the default windowing system used in UNIX?

X Windows

1.72. What command is used to list the contents of a directory?

ls

1.73. What command is used to display a list of currently running processes?

ps

1.74. What is a login shell?

The login shell is the program that is executed when a user logs into the system. This program accepts user commands and executes them. This is the main user interface to the operating system. Examples of shells include bash, sh, tcsh, csh and ksh.

1.75. What is a UID?

UID stands for User Identification. It is the unique number that is assigned to every user on a UNIX system.

1.76. In what file is the relationship of UID to username defined?

/etc/passwd

1.77. What command is used to check a filesystem for errors?

fsck

1.78. Is there a difference between a file called OUTPUT.TXT (in all caps) and output.txt (all lowercase) in UNIX?

Yes. UNIX operating systems are case sensitive.

1.79. What command is used to read the manual page for a given command?

man <command>

1.80. What symbol is used to redirect standard out (STDOUT) to a file?

> (the greater-than sign)

1.81. What command is used to establish a secure command-line session with another system over a TCP/IP network?

ssh

1.82. What file contains the list of drives that are mounted at boot?

Solaris: /etc/vfstab
Other UNIX: /etc/fstab

1.83. In what file is the default runlevel defined?

/etc/inittab

1.84. Define and describe TCP Wrappers. What are the reasons for using TCP Wrappers in addition to a host-based firewall?

TCP Wrappers is a tool used to restrict inbound access to network services. Using the files /etc/hosts.deny and /etc/hosts.allow, a system administrator can define what range of IP addresses or hosts are allowed or denied access to certain ports on a system. TCP Wrappers does not replace a firewall! For one thing, TCP Wrappers will only protect those services that query it to control access. A common program that does this is inetd, the internet superserver. So why use TCP Wrappers in addition to a firewall? TCP Wrappers provide a second line of defense in the event the firewall is compromised. TCP Wrappers also offer excellent logging capabilities, giving you a way to verify your firewall logs.

1.85. Define and discuss the security ramifications of each of these situations: Current directory (“.”) in root’s $PATH variable, “r” commands (rlogin, rcp, etc), setUID bit on /bin/vi

If an administrator is coming from the DOS or Windows world, he is probably surprised when he tries to execute a binary in his current working directory and he receives the error “command not found”. Unlike DOS and Windows, the current working directory is not in a user’s $PATH by default. This means that in order to run the program /home/bin/program, you must either type the full path name at the command line, cd to /home/bin and type “./program”, or add /home/bin/ to your $PATH variable.

It is possible to replicate this windows functionality on UNIX systems. In UNIX, the current working directory can be referred to with the period (“.”). To make the UNIX command line work like the Windows one, you could issue the command “PATH=.:$PATH”. This will set the current

directory to be the first place that the shell searches for binaries when you issue a command.

This, however, is an enormous security risk. In a multi-user environment like UNIX, you must always be aware that other people are using the same system. If a root user were to alter his $PATH this way, a malicious user could create a file in the /tmp directory called ‘ls’. This file could be a shell script that could do anything at all, even deleting the entire system. If this is made executable, and the root user changed directories to /tmp and issued the ‘ls’ command, the malicious script is executed (as root) instead of the system /bin/ls.

The “r” commands have been popular for many years on UNIX systems. These commands exist to give remote access to users using multiple UNIX systems.

There are two inherent flaws with these commands that make them essentially useless in a secure networking environment. First, these command all transmit data over the network in plain text. Anyone running a packet sniffer that has access to the same network segment will see all of the data being transmitted. Second, the authentication system for these commands is based on restricting and allowing access by IP address only. It is far too easy to spoof an IP address, meaning that unauthorized access to these services is relatively simple.

In order to achieve the same functionality that these commands offer, but to do it in a secure way, install the OpenSSH software suite. This package is an open-source implementation of the ssh (secure shell) applications and other similar applications. Not only does ssh encrypt all data that is sent over the network, it supports more advanced authentication mechanisms, such as public/private key based authentication.

It is very important that the setUID bit be used with care. The UNIX permissions model is based on a parent-child relationship. When a parent process spawns a child process, the child process inherits the permissions of the parent. For example, if an application gives the user an option of spawning a shell, that applications can never be assigned the setUID bit. If

it were, the shell that it spawned would have the permissions of root, giving any user full access to the system. In the same vein, an application like the vi editor can never be setUID, since it would then allow any user to modify any file on the system, completely bypassing the UNIX security model.

1.86. What is the advantage of giving a user elevated privileges to certain commands through sudo, as opposed to giving them root access to a system?

With the sudo command, it is easy to limit what command a user may run as root through entries in the /etc/sudoers file. This allows a system administrator to create varying levels of access for certain users, without having to give out the root password. Sudo also logs every attempted access and every command that it runs, leaving a detailed audit trail behind for all programs run with superuser privileges.

1.87. What is the chroot command and why is it important to the security of a system?

The chroot command stands for 'change root'. This command allows a system administrator to set up a restricted section of the filesystem for users to use (commonly called a 'chroot jail'). This filesystem appears to the user to be the complete filesystem, but in reality it is merely a subdirectory of the 'real' filesystem, containing just those commands and directories that the system administrator wishes the user to have access to.

Although the UNIX permissions model is a tried and true one, it is still fraught with enough complications that few system administrators would feel comfortable with strange users roaming through their machines. Chroot gives them the ability to grant access to certain users, but to limit that access in such a restrictive way that the user cannot access something the administrator does not want them to access.

Chroot is also commonly used when running system services. Most UNIX system services run as the root user. This makes these services extremely vulnerable to buffer overflow attacks. If a "buffer overflow attack" succeeds, the attacker will be left with a shell on the system that has inherited the permissions of the service owner (in this case, root). However,

if the service is run in its own chroot jail, the damage caused by these kinds of attacks is greatly minimized, because the attacker is left with an extremely limited shell as opposed to their entire filesystem.

1.88. Describe a buffer overflow attack and list possible defenses.

A buffer overflow attack is perhaps the most common system compromise in the history of computing security. When a programmer writes software that expects input, that expected input is assigned a buffer in the system. The size of that buffer is dependent upon the expected size of the data. If the programmer has not put checks in place to validate the size of the input data, a malicious user can send more data than can be allocated in the buffer, causing an overflow. In the best case, this will cause memory corruption in the program and it will fail. In the worst case, specific data can be sent in the overflow portion of the input data, which is then executed with the privileges of the owner of the program. In the case of a system service running as root, a remote buffer overflow attack can allow an attacker to run arbitrary code as the root user on a system.

The real defense against a buffer overflow attack is to prevent them from ever happening. This can be accomplished through a complete code audit, ensuring that every data input point in an application does proper bounds checking to ensure that only the amount of data that can be allocated is accepted. In practice, this is often difficult to do, even with the available code profiling tools available.

A common practice is to treat this kind of attack like any other in the networking security realm - assume it will happen and make plans to identify it and reduce the damage. Identification is normally handled by examining the logs of a system, looking for strange log entries from a system service. If a system has been compromised, file system integrity tools will tell you what files on the system have been modified. A “chroot jail” is commonly used to minimize the amount of damage that is possible from an attack of this nature. This has become a common enough solution that some popular server programs have built-in chroot functionality (sendmail is an example).

In Linux, there are kernel modules available to assist in the defense against buffer overflow attacks. These patches go about this in different ways, but most focus on preventing arbitrary code from being inserted onto the stack and executed, making traditional buffer overflow attacks much more difficult to execute.

1.89. What is the weakest point in any network or system's security? What can you do about it?

People. No matter how much money you have spent on the latest in firewalls, intrusion detection tools, forensic analysis and physical security, people are always the weakest link. This is a fundamental rule of computer security that everyone who works in the field needs to realize. Simply having an easy-to-guess password can completely compromise the security infrastructure of an entire organization.

What is the answer to this problem? Education. An educated user is much less of a security risk. Security is a tricky thing when it comes to users: be too lenient, and they will ignore it. Be too stringent, and they will do everything in their power to bypass is. Finding the right balance is a challenge. The first step to this is educating your work force. Telling a user why the company's password policy should be followed, instead of threatening to fire them if they disobey, you will probably get a better response. Security needs to be a standard part of everyone's job. In this day and age, everyone is responsible for the security of a company's systems and datBy letting a user know what is acceptable and what isn't, you will go a long way towards making your environment a much more secure place.

1.90. Describe a file system integrity tool and explain its importance.

A file system integrity tool is an application that maintains a record of all the files on a system. This record will contain every bit of data about each file, including time stamp, size, owner, location, etc. Periodically, a check of the existing file system against this record is run. If part of the filesystem has been modified, it will not match this record, and the system administrator will be notified.

This is an important tool because YOUR SYSTEMS WILL GET BROKEN INTO! It is not a question of IF; it is only a question of WHEN. Assuming that you alone have an infallible security model is the equivalent of refusing to face reality. Due to some error, either human, hardware, or software, your systems will be compromised. The largest problem that arises from this event is the time it takes the system administrator to figure out that a compromise took place. Stories abound of companies finding out months later that they were compromised, after the attacker stole all of their data and used their systems to attack other systems on the internet.

A file system integrity tool will notify you if a part of your system has been modified. It is the first step in recovering from an intrusion.

1.91. What is an intrusion detection system and what are some basic flaws in the concept of 'detecting intrusion'?

An "intrusion detection system" (IDS) is software that monitors a system or network, looking for evidence of activity that can be identified as malicious.

There are a number of different kinds of IDS systems available. The simplest of these will monitor system log files looking for any suspicious messages generated by the system services. A second type of IDS will monitor the activity of a network or system and match what it finds against a pattern of known 'attack signatures'. Similar to anti-virus programs, these IDS systems are able to identify an intrusion if it follows the pattern of a previously known attack. The third kind of IDS system is also similar to an anti-virus program. Many anti-virus programs use heuristic analysis to identify programs that 'look like' viruses. In this same way, an IDS can identify system activity that appears to be malicious but does not match any known malicious pattern.

The inherent flaw in all of these systems is that it is very difficult to write an algorithm that can differentiate between 'good' and 'bad' activity. Since these systems do not take into account the context of a pattern, there can be many false positives. An IDS system that generates too many false positives is worse than no IDS system at all. A system administrator will simply start to delete all of the notifications while believing that his network and systems are secure.

Although flawed, when configured correctly and kept up to date, IDS systems can be an important piece in the overall security plan of an organization.

1.92. What is “ARP cache poisoning”, and how can it be prevented?

The ARP (Address Resolution Protocol) cache on a system is what translates IP addresses to MAC addresses on a local TCP/IP network. It is relatively easy to spoof ARP replies, meaning that if one system on your network is compromised, that system can spoof other systems on the network easily. Your server will think that it is connecting to a trusted host, when in reality it is connecting to the compromised system, or even worse, another host on the Internet.

To prevent this from happening, you can define a list of systems that your server must trust. In the file /etc/ethers, you can hard code the MAC address to IP address mapping of these systems, preventing ARP from overriding this information. This disadvantage of this approach is that if the network card changes in any of these systems, the MAC address will change as well, meaning this file must be updated.

1.93. Describe VPN technology and list some common implementations.

A VPN (Virtual Private Network) is an encrypted tunnel that exists to transmit data over an otherwise insecure public network. VPNs are most commonly used to transmit secure data over the internet. Instead of handling the encryption and authentication at the application layer (like SSL enabled web sites), a VPN handles all encryption at the network layer, allowing traditional networking protocols to use the VPN tunnel without modification.

There are three common VPN solutions in use today:

- IPSec based solutions. IPSec is the security protocol built into the next version of IP, Ipv6. Many vendors have standardized on IPSec as their VPN solution, including Cisco and Sun Microsystems.

- PPTP (Point to Point Tunneling Protocol) based solutions. The Point to Point Tunneling Protocol is a modification to the standard PPP protocol which allows for encryption and authentication at the network layer. PPTP support is built into later version of Microsoft Windows, and implementations are available for most UNIX operating systems.
- Proprietary solutions. Since a VPN tunnel is essentially just encrypted traffic, any reliable encryption algorithm can be used to create a tunnel. It is even possible to use the encryption functionality of the ssh application to build a VPN tunnel.

1.94. What is a DMZ and why would you want one?

"DMZ" stands for "Demilitarized Zone". This is a term that describes a small network that exists between a corporate network and the internet. The DMZ will house publicly available servers, such as web servers and mail servers. Users inside the company network can access hosts in the DMZ, and users outside the company network can access hosts inside the DMZ, but the DMZ acts as a buffer between the company network and the internet. If a host in the DMZ is compromised, the attacker will not gain access to any server inside the company.

1.95. On "Linux", how can the directory entries at the top of the /proc filesystem aid in security?

When a system is compromised, a rootkit is commonly installed by the intruder. A rootkit is a set of trojaned programs that replace common system programs. An example would be a trojaned version of the 'login' program which always accepts a certain username/password combination. Another example is a trojaned version of the 'ps' program which will display all running processes except the ones that the attacker wished to hide.

The first step in security is knowledge. What is running on your system? Traditionally, the 'ps' command is used to display this information. But what if you don't trust the 'ps' command? In this case, you need a deeper understanding of how the operating system works. In Linux, all of the informational commands are simply frontends for data that is available as text files or directories in the /proc filesystem. Specifically, every running

process on a Linux system will have a directory under /proc. For example, the init process (PID #1), will have a directory of /proc/1. So if you are ever in doubt about the reliability of the 'ps' command, compare its output with the directories in /proc. If you see a discrepancy, you probably have a compromised system.

The moral of this story is that there is no such thing as too much knowledge. By having a good understanding of how things work behind the scenes, you are better able to handle unexpected situations.

1.96. What command is used to display the ports that are open on a UNIX system?

The "netstat" command will display all open ports and all network connections.

1.97. What is the subnet mask for a default Class-C IP network?

255.255.255.0

1.98. In a standard class C IP network, how many IP addresses can be assigned to hosts? What are the other non-assignable addresses used for?

A Class C network has a total of 256 (28) IP addresses. However, only 254 of these can be assigned to hosts. The first IP address in the subnet is the network IP address, which is used to refer to the network as a whole. The last IP address in the subnet is the broadcast address, which is used to communicate with each host on the network.

1.99. What is a MAC address?

A "MAC" (Media Access Control) address is a 48-bit address that is assigned to every hardware device that is designed to communicate on an Ethernet network. Hardware manufacturers are assigned a range of MAC addresses to use for their products to ensure uniqueness.

1.100. You wish to set up a Linux system as a router between two subnets. You have installed and configured two network cards,

each attached to a different subnet. What are the steps that need to occur before the system will act as a router?

Step 1: You must configure the packet forwarding rules. In recent versions of the Linux kernel (2.4 and later) this is accomplished with the 'iptables' command.
Step 2: You must instruct the kernel to forward packets between the two interfaces. This is accomplished with this command:
echo 1 > /proc/sys/net/ipv4/ip_forward

1.101. What is an MTU?

The MTU (Maximum Transmission Unit) is the largest packet size that can be sent over a TCP/IP network. If a packet is larger than the MTU of the operating system or router, it will get broken into smaller packets. You can often increase network performance by ensuring that your operating system is using the same size MTU as your local router.

1.102. What file determines the DNS servers that are queried when a network request is made?

/etc/resolv.conf

1.103. What is the difference between UDP and TCP?

UDP and TCP are both Layer 4 (Transport) protocols in the IP protocol stack. UDP (User Datagram Protocol) is considered an 'unreliable' protocol because it does no end-to-end verification or packet numbering during transmission. The higher-level application that utilizes UDP must be responsible for the reassembly of the UDP packets. Examples of UDP based services are tftp and NFS.

TCP (Transmission Control Protocol) is considered a reliable protocol because it has built in tools for guaranteeing delivery of packets and a system for requesting the resending of a packet if packet loss occurs. Because of this, TCP incurs more overhead then UDP. Examples of TCP based services are SMTP and HTTP.

1.104. What is the difference between POP and IMAP?

POP stands for Post Office Protocol and is a simple set of instructions that let your computer retrieve messages from a mail server. After authenticating the request with the user's password, the POP protocol asks the server if there is any new mail and if your mailbox on the server has messages, then POP usually downloads them to your computer and deletes them off the server.

IMAP stands for Internet Mail Access Protocol and allows a "client" email program to access a message stored at a remote location as if they were on the local computer. Email stored on an IMAP server can be manipulated from a desktop computer at home, a workstation at the office, or a notebook computer while traveling, without the need to transfer messages or files back and forth between these computers.

Generally, you want to use POP if you access your mail from one computer, and IMAP if you need to manipulate your mail from more than one computer, say an office and a home computer. Below is a table to help you understand which service fits you better.

POP service:

- POP was designed for, and works best in, the situation where you use only a single desktop computer.
- Normally, messages are downloaded to your desktop computer and then deleted from the mail server.
- If you choose to work with your POP mail on more than one machine, you may have trouble with email messages getting downloaded on one machine that you need to work with on another machine; for example, you may need a message at work that was downloaded to your machine at home.
- If you choose the POP option "keep mail on server", your POP "inbox" can grow large and unwieldy, and email operations can become inefficient and time-consuming.
- Your archive of mail, if you have one, is kept on your desktop computer - you generally need little storage space on the mail server.

IMAP service:

- IMAP is designed for the situation where you need to work with your email from multiple computers, such as your workstation at work, your desktop computer at home, or a laptop computer while traveling.
- Messages are displayed on your local computer but are kept and stored on the mail server - you can work with all your mail, old and new, from any computer connected to the internet.
- You can create subfolders on the mail server to organize the mail you want to keep.

1.105. How do I know if I should use IMAP or POP?

The table below may help you decide which email protocol is better for you to choose.

Choose IMAP if:

- you need access to your email, new and old, at multiple locations, e.g., office, lab, home.
- you use POP, and specify to leave your mail on the server.
- you use the option to synchronize the emails on your email client and the server.

Choose POP if:

- you work with your email in a single location or on a single computer.
- Webmail works fine for you when you need to check new email remotely.
- you regularly backup your email messages and archive it.

1.106. Describe Simple Mail Transfer Protocol.

SMTP (Simple Mail Transfer Protocol) is a TCP/IP protocol used in sending and receiving e-mail. However, since it's limited in its ability to queue messages at the receiving end, it's usually used with one of two other protocols, POP3 or Internet Message Access Protocol, that let the user save messages in a server mailbox and download them periodically from the server. In other words, users typically use a program that uses SMTP for sending e-mail and either POP3 or IMAP for receiving messages that have been received for them at their local server. Most mail programs such as

Eudora let you specify both an SMTP server and a POP server. On UNIX-based systems, sendmail is the most widely-used SMTP server for e-mail. A commercial package, Sendmail, includes a POP3 server and also comes in a version for Windows NT.
SMTP usually is implemented to operate over Transmission Control Protocol port 25.

Simple Mail Transfer Protocol (SMTP), documented in RFC 821, is Internet's standard host-to-host mail transport protocol and traditionally operates over TCP, port 25. In other words, a UNIX user can type telnet hostname 25 and connect with an SMTP server, if one is present.

SMTP uses a style of asymmetric request-response protocol popular in the early 1980s, and still seen occasionally, most often in mail protocols. The protocol is designed to be equally useful to either a computer or a human, though not too forgiving of the human. From the server's viewpoint, a clear set of commands is provided and well-documented in the RFC. For the human, all the commands are clearly terminated by newlines and a HELP command lists all of them. From the sender's viewpoint, the command replies always take the form of text lines, each starting with a three-digit code identifying the result of the operation, a continuation character to indicate another lines following, and then arbitrary text information designed to be informative to a human.

If mail delivery fails, sendmail (the most important SMTP implementation) will queue mail messages and retry delivery later. However, a backoff algorithm is used, and no mechanism exists to poll all Internet hosts for mail, nor does SMTP provide any mailbox facility, or any special features beyond mail transport. For these reasons, SMTP isn't a good choice for hosts situated behind highly unpredictable lines (like modems). A better-connected host can be designated as a DNS mail exchanger, then arrange for a relay scheme. Currently, there two main configurations that can be used. One is to configure POP mailboxes and a POP server on the exchange host, and let all users use POP-enabled mail clients. The other possibility is to arrange for a periodic SMTP mail transfer from the exchange host to another, local SMTP exchange host which has been queuing all the

outbound mail. Of course, since this solution does not allow full-time Internet access, it is not too preferred.

RFC 1869 defined the capability for SMTP service extensions, creating Extended SMTP, or ESMTP. ESMTP is by definition extensible, allowing new service extensions to be defined and registered with IANA. Probably the most important extension currently available is Delivery Status Notification (DSN), defined in RFC 1891.

1.107. What steps are necessary to configure a Solaris system to participate in an IP network?

This assumes that the Solaris system is not part of an NIS/NIS+ network.
1 /etc/hosts must contain the hostname and IP address of the system.
2 /etc/hostname.<interface> must contain the hostname of the system. <interface> is the corresponding interface; for example,
3 /etc/hostname.dmfe0
4 /etc/netmasks must contain the subnet mask for the system.
5 If using DNS, /etc/resolv.conf needs to contain the primary DNS server.
6 /etc/defaultrouter must contain the IP address of the router for the subnet.

1.108. What is TCP sequence number prediction and what can you do to prevent it?

Since TCP is a reliable protocol, each TCP packet is assigned a sequence number. It's the responsibility of the receiving end of a TCP connection to put the incoming packets in order, and re-request a packet if one got lost along the way. Theoretically, if a third system can successfully trick one of the two systems in a TCP conversation into communicating with it instead of the original system, the third system can break into the conversation. This is known as 'session hijacking', or the 'man-in-the-middle' attack. In order for this attack to work, the third system has to know the TCP sequencing being used between the original systems. If this sequencing is based on a known algorithm, determining the next TCP sequence number is trivial and this attack becomes easier.

This vulnerability didn't come to light until around the year 2000. At that time, most UNIX vendors issued patches to their kernels to ensure that the algorithm used to calculate TCP sequence numbers was more complicated. There are, however, many systems on the internet today that are still vulnerable to this kind of attack. The moral of the story is: keep your systems updated.

1.109. You have purchased a domain name and setup one of your UNIX servers as a DNS server. What entry must you make in your primary zone file to ensure that email addressed to your domain is sent to the correct mail server?

The MX record is the DNS entry that is responsible for defining the mail server for a particular domain. It is recommended that a domain have multiple MX records for the sake of redundancy.

1.110. Associate these common network services with their default port numbers: Telnet, SSH, FTP, SMTP, DNS, POP3, HTTP, HTTPS, IMAP

Obviously, one is not expected to memorize all of the common port numbers and services. However, these are common enough that all good UNIX system administrators should know them by heart.

Telnet: TCP 23
SSH: TCP 22
FTP: TCP 20 & TCP 21
SMTP: TCP 25
DNS: TCP 53 and UDP 53
POP3: TCP 110
HTTP: TCP 80
HTTPS: TCP 443
IMAP: TCP 143

1.111. What command will display the routing table on a UNIX system?

netstat -r

1.112. Your UNIX system is able to ping other devices on the local subnet, but it is unable to ping a device outside of the subnet. What could be the problem?

In order for a host to communicate outside of its subnet, it must know the default router address. This error could indicate that the default router is not set or is incorrect. If the default router setting on the host is correct, then there may be a problem with the router itself.

1.113. When you initiate a telnet session to your UNIX system, the connection is established, but there is a 20 second or more pause before you see a login prompt. What could cause this?

The telnet server performs a reverse DNS query on every IP address that attempts a connection. If DNS is configured incorrectly, or the DNS for the network is down, the server will wait for a DNS timeout before presenting a login prompt to the telnet client.

1.114. Describe the default partitioning scheme in Solaris, denoting the slice number, the default mount point, and the purpose.

1 Slice 0 is mounted as the root partition (/) by default. This is the top level UNIX file system.
2 Slice 1 is reserved for swap space. The size of this will depend upon the amount of memory and the duties the system will perform.
3 Slice 2 represents the entire disk and is not mounted.
4 Slice 3 is mounted as /export by default. Commonly used to hold exported information, such as user home directories or alternative versions of operating system binaries.
5 Slice 4 is mounted as /export/swap by default. This partition is used to provide virtual memory space for client systems. Slice 4 is also commonly mounted as /opt.
6 Slices 5 is mounted as /opt by default. This is where packages are installed that are not critical to the functioning of Solaris. If /opt was mounted already as Slice 4, this slice is often mounted as /var.
7 Slice 6 is mounted as /usr by default. /usr contains binaries and libraries that are used by normal system users. /usr is commonly shared between systems.

8 Slice 7 is mounted as /home, /export/home or /export/spare by default. Commonly holds user directories or other shared information.

1.115. Define and describe the following terms related to disk management under UNIX: Boot block, Super block, Logical block, Inode

The boot block (or master boot record on Intel-based systems) stores the information needed to boot the system. The data starts at sector (or cylinder group) 0 and can reside on any disk that the hardware is capable of booting from. For example, if you're using Linux on an x86 based system, and your system BIOS supports booting off of a USB drive, you can use a removable USB drive as your boot device.

The super block contains information about the file system itself. This information includes:

1. file system size and status
2. file system label
3. file system state (clean, dirty)
4. logical block size

Without the superblock, the file system is unusable. Because of its importance, when a file system is created on a disk, multiple copies of the superblock are created on different parts of the disk for use in case the main superblock becomes corrupted.

A logical block is the smallest unit of data that can be stored on a file system. The size of the logical block on a particular file system is determined when that file system is created. Common sizes for logical blocks are 8K, 4K, 2K, 1K and 512 bytes. If you save a file that is smaller than the logical block size, the file will still take up one logical block of space.

An inode (or index node) contains all of the information about a file except the filename itself. Every file on a UNIX system has an inode, containing information such as:

1 Type of file (regular, directory, link, character special, etc)
2 Last access time, last modified time, creation time

3 Size of the file
4 Permissions and ownership of the file

The number of inodes allocated to a file system is fixed at the time the file system is created. If this number is exceeded, no new files can be created on the file system, even if there is free space available.

1.116. Describe the processes for adding a new hard disk to a UNIX system.

Step 1: Physically install the hard disk in the machine. If it is an IDE disk, ensure that the proper master/slave jumper settings are set. If it is a SCSI disk, ensure that a unique LUN is used and that termination on the SCSI chain is correct.
Step 2: Partition the disk. Use the 'format' command on Solaris or the 'fdisk' command on Linux or BSD systems.
Step 3: Create a file system on the disk. On Solaris and BSD, use the 'newfs' command. On Linux, use the 'mkfs' command appropriate for the file system type (ie mke2fs, mkreiserfs, etc).
Step 4: If it doesn't already exist, create a mountpoint for the new disk. mkdir /mnt/newdisk
Step 5: Ensure that the disk mounts automatically when the system boots. On Solaris, add an entry to /etc/vfstab. On BSD and Linux, add an entry to /etc/fstab.
Step 6: Mount the file system. mount /mnt/newdisk

1.117. What is the difference between a regular file system and a journaling file system?

In today's world, all file systems take advantage of a write cache, meaning that requests to write to the disk are actually written to memory first. The kernel then decides when to flush this cache to the disk (normally when there is some level of system downtime). This speeds up the overall performance of the system. The problem with this method arises when a disk is unmounted or becomes unavailable before the cache is flushed. This can happen if power is lost or a system failure occurs. When this happens, before the disk can be mounted again, it must be scanned in order to analyze the disk and bring it back to a consistent state. With large disks, this

can take a significant amount of time, during which the system is unavailable.

In order to address this problem, journaling file systems were developed. By borrowing a technology present in the database world for many years (transaction logs), a file system is able to recover much more quickly from an unplanned failure. Instead of scanning the entire disk, only the transaction log is examined, reducing the recovery time from possibly hours to seconds.

1.118. Define and describe the common UNIX file systems.

- UFS – The UNIX file system. The original standard for UNIX file systems and the default on BSD and Solaris, it is based on the original BSD FAT Fast file system.
- JFS – Journaling file system. This file system was developed by IBM and is the default file system under AIX. IBM has also ported this file system to Linux.
- EXT2 – The standard Linux file system. This is a non-journaling file system.
- EXT3– EXT2 with journaling capabilities.
- XFS – Silicon Graphics' journaling file system. XFS is the default file system in IRIX and has also been ported to Linux.
- ReiserFS – Journaling file system developed by Hans Reiser. Touted for its speed and scalability. Available in most Linux distributions.

1.119. What are the 3 settings in the standard UNIX file permissions model, and how does each apply to files and directories?

For files:
Read – Can open the file and read its contents
Write – Can modify the file
Execute – Can execute the file

For directories:
Read – Can list the files in a directory
Write – Can add or removes file from the directory

Execute – Can execute a program within the current contents of the directory

Directory permissions are a little non-intuitive. For example, it's not immediately obvious what the 'execute' option on a directory would give you, since you can't actually execute a directory. It does not mean that you can execute binaries that are inside that directory. If you do not have execute permissions on a directory, you will not, for example, be able to list the contents of that directory. Because the 'ls' command needs to run against the directory in question, the command 'ls <directory>' will be denied.

A frequently asked question about UNIX permissions is "How do I grant the 'change' option to a user?" UNIX doesn't have a 'change' bit, but you can replicate the functionality within the existing permissions model. 'Change' means that you can't delete a file, but you can modify it. To replicate this in UNIX, you first ensure that the user has write access to the file itself. This allows them to modify the file. Then, to ensure that they can't remove it, revoke their write privileges on the directory that houses the file. This will prevent them from removing the file itself.

1.120. What is the setUID bit and how is it employed?

When the setUID bit is enabled for an executable file, that file will execute with the permissions of the file owner rather than the permissions of the parent process. This is useful for enabling regular users to run privileged commands without having to give them root access. setUID is set with the chmod command. To change the permissions on an executable file so the setUID bit is set, owner has read, write and execute, group has read, write and other has read and write, issue the following command:

chmod 4755 <filename>

1.121. Define and describe the cron job scheduling system. What are the fields available to define the time a job will execute?

Cron is the default UNIX job scheduler. Cron jobs are defined per user in /var/spool/cron/<username>. This file lists each user's cron jobs, one per

line. Each line contains 6 fields:

- Field 1 – Minute
- Field 2 – Hour
- Field 3 – Day of the month
- Field 4 – Month
- Field 5 – Day of the week
- Field 6 – Program to run

By default, any output sent to STDOUT or STDERR from these scheduled programs is emailed to the user.
How do you disable a user account without deleting it? Assume shadow passwords are being used.
The command 'passwd -l <username>' will disable an account. It does this by placing an "!" in front of the user's encrypted password in the /etc/shadow file.

1.122. In the standard lpr-based UNIX printing system, what command will display the status of a print queue?

lpq - that is a spool queue examination program.

1.123. Describe the Samba suite of utilities. How are they useful in a heterogeneous environment?

Samba is a software suite that implements the Server Message Block protocol on UNIX systems. This is the protocol that Microsoft Windows systems use for sharing filesharing, printer sharing, and authentication. The Samba server software allows a UNIX machine to act as a file, print, and authentication server for Windows clients. The Samba client software allows a UNIX machine to participate in a Windows network, accessing shared drives, printing to shared printers, and authenticating against a domain controller or an Active Directory server.

1.124. Describe the package management options available on Linux systems.

Most Linux distributions utilize either the rpm or the deb package format. Rpm based distributions include RedHat, SuSE and Mandrake. The Debian -based distributions utilize the deb package format, including Debian itself,

Knoppix, Lindows and Xandros Linux. Both systems offer dependency checking, allowing for a level of intelligence when installing or upgrading software.

1.125. Describe the package management software on Solaris.

The pkgadd command is used to install software. When you receive a package from Sun, or download one from the internet, you install it with the 'pkgadd -d <package name>' command. To see a list of packages installed on your Solaris system, run the command 'pkginfo'. To list the files that belong to an installed package, run the command 'pkgchk -l <package name>'.

1.126. What are three ways to find a file on a UNIX system?

The 'which' command will display the full path to a command if that command is in a directory defined by your $PATH environment variable. The 'locate' command will query a hash database containing a list of all files on the filesystem. The database must be updated periodically. This command is not a standard part of all UNIX systems. The 'find' command can search entire partitions for a file, using an exact or regular expression match.

1.127. If /etc/inittab defines your default runlevel as 3, where do your system startup scripts reside?

/etc/rc3.d

1.128. You are logged in as root and wish to kill a process with PID #1054. You type 'kill 1054' but you still see the process in the process table. Why did the process not die, and what can you do to get rid of it?

When called with no arguments, the kill command by itself sends the TERM signal to a process. The TERM signal can be best described as 'asking the process to end when it's ready'. A process can be configured to specifically catch the TERM signal and ignore it. The process could also be in such a state that it is unable to respond to the TERM request. In this case, more drastic measures are called for. By passing the -9 option to kill ('kill -9 1054'), you are sending the KILL signal. This is a lower level signal that

tells the kernel to terminate this process immediately. This is usually reserved for processes that are locked in some loop or wait state and do not heed the TERM signal. If a process does not respond to kill -9, it is usually in some non-recoverable state (like waiting on a disk that is no longer available). The only way to eliminate these processes is to reboot the system.

The heart of the UNIX operating system is the command line. Having a good grasp of what commands are available is fundamental for working with a UNIX OS. In the same vein, since everything in UNIX is a file, knowledge of the file system is crucial. These questions can quickly weed out an experienced UNIX sysadmin from a trainee.

1.129. What are the fields defined in the /etc/passwd file?

- Field 1: username
- Field 2: placeholder for encrypted password (assuming shadow passwords are being used)
- Field 3: UID (User Identification)
- Field 4: GID (Group Identification)
- Field 5: Comment
- Field 6: Home directory
- Field 7: Login shell

1.130. What is the purpose of the ‘grep’ command?

grep is used to search for strings of text in files. Grep stands for “Global Regular Expression Print”

1.131. What command is used to display information about the network interfaces on a system?

ifconfig

1.132. The ‘ps’ command lists processes currently running on the system. Identify and describe the following column headers from the output of ‘ps -aef’: UID, PID, PPID, TTY

UID = User Identification, the owner of the process
PID = Process ID. Each UNIX process is assigned a unique process ID.

PPID = Parent Process ID. The PID of the parent of this process.
TTY = On what terminal is this process running. If the TTY is "?", then the process is not running on a terminal (either a system process or a detached process).

1.133. In Solaris, what command is used to display the slice information for a particular disk?

prtvtoc <device>

1.134. What command is used to display the disk space occupied by mounted file systems?

df

1.135. What file contains a list of currently mounted file systems?

/etc/mtab (on Linux)
/etc/mnttab (on Solaris)

1.136. On Solaris, what file contains settings that define the default maximum time a password is valid and the default minimum time period before a password must be changed?

/etc/default/passwd

1.137. What command and options would you issue to set the permissions on a file to read, write, execute for owner, read and execute for group, and no permissions for other?

chmod 750 <filename>

1.138. What command will display the default file permissions that will be assigned when a user creates a file?

umask

1.139. On Solaris, what command is used to display the "Access Control List" entries for a file?

getfacl <filename>

1.140. What command is used to run an application with a different security context than the user's default? What is the configuration file for this command, where programs are listed which defined users can execute with higher privileges?

sudo <program>
/etc/sudoers

1.141. What file defines the default runlevel of the system and dictates what directory holds the startup scripts for that runlevel?

/etc/inittab

1.142. What command is used to check a filesystem for errors?

fsck

1.143. Why are there normally two bin directories, /bin and /usr/bin?

The UNIX filesystem was designed to be shared by multiple systems. In a shared environment, it is unnecessary to have data duplication across multiple systems. The /usr partition on a UNIX system is designed to be shared by clients. This is normally accomplished by client systems mounting a server's /usr partition as their own. This way, all clients have access to the same programs without a system administrator having to maintain copies on multiple systems.

Because of this fact, files in the /usr partition must never be considered critical system files. If a client needs to boot up off of the network for some reason, they will not have access to their /usr partition if it is mounted from a central server. If some sort of system repair ability is necessary on a client system, that program needs to live in the /bin directory. By convention, the /bin directory is local to every system.

So the reason there are two bin directories is that /bin contains local system-level binaries that are critical for the correct operation of the system, while

/usr/bin contains binaries that are less important and not critical to the basic functioning of the system.

1.144. On Solaris, what file defines the master kernel configuration file?

/etc/system

1.145. At the Solaris OpenBoot prompt, what command will display all of the SCSI devices connected to the system?

probe-scsi

1.146. How do you boot a Solaris system into single user mode?

1. Get to the OpenBoot prompt by hitting Stop-A
2. Boot the system with the command 'boot -s'

1.147. What command allows you display and to modify your keyboard mappings in X Windows?

xmodmap

1.148. In Linux, what file contains a list of all IRQs currently in use?

/proc/interrupts

1.149. You attempt to unmount a currently mounted disk and you receive the error "Unable to umount: device or resource busy". What does this mean and what must you do before you can unmount the disk?

This means that some process is currently accessing the file system you are trying to unmount, either through an open file handle or having a process with a current working directory in the file system. You first need to identify what processes are accessing the file system and who owns those processes. You then must decide how to deal with those processes.

The 'fuser' command will display the processes that are accessing a given file system. After you have determined what the processes are and who owns them, you can either 1) wait until those processes complete before

you attempt to unmount the disk, 2) contact the user(s) in question to determine the status of the processes, or 3) kill the processes and unmount the disk.

1.150. You need to run the fsck program against a file system, but you must do this while the file system is unmounted. Since the file system in question is the root file system, you cannot unmount it while the system is running. What options do you have for running fsck against this file system?

There are two ways to get fsck to run against this file system when it is not mounted.

Access the disk from a different root partition. This can be done be either removing the disk itself and mounting it under another UNIX system, or booting your system with a root or rescue disk. Either way, you have access to the fsck command without relying on your original disk being mounted. You can then fsck the drive and return the system to its previous state.

Issue the command 'shutdown -r -F now'. The '-F' option to shutdown forces the system to run fsck against the drives defined in /etc/fstab (or /etc/vsftab) before they are mounted on the next reboot. The '-r' options tells shutdown to reboot the computer immediately.

1.151. You have forgotten root's password. What options do you have to get back into the system as root?

This answer will be higher-level to cover most flavors of UNIX; individual implementations will vary depending upon the UNIX flavor.

Root's password is normally stored in encrypted form in either /etc/passwd or /etc/shadow. While it is impossible to recover what the password was, you can change the password, or reset it to a null value.

The first step is to access the disk containing the root partition without booting to it. You can do this by either booting to a root rescue disk and then mounting the original root partition, or removing the drive itself and

mounting it in another system. By accessing the root file system this way, you are bypassing any of the security on the file system itself.

When you have access to the root partition, locate the encrypted password string for root in /etc/passwd or /etc/shadow. Erase the encrypted string. Save your changes and return the system to its normal configuration (if you moved the disk, return it to the original system. If you booted off of a different boot disk, remove it and boot off of your original disk).

After rebooting, you should now be able to log in as root without giving a password. You can then reset root's password with the 'passwd' command.

1.152. Your UNIX workstation is unable to ping any hosts on the local TCP/IP network. What steps do you take to troubleshoot this problem?

This kind of question is a very revealing one. How a candidate answers this questions will determine their thought process, specifically how they approach a problem from a logical standpoint. Ideally, you want a candidate to display knowledge of what settings are necessary to enable communication on an IP network, but more importantly, you want them to follow the scientific method in diagnosing and resolving the problem.

The following items need to be correct in order to communicate on a local TCP/IP network:

- Some physical connection to the network, whether it be wired or wireless
- An IP address that is part of the local subnet, but is unique to that subnet
- A subnet mask that correctly reflects the configuration of the subnet
- A correct gateway IP address
- If name resolution is desired, a correct name server must be configured

From a troubleshooting standpoint, it's always best to start at the lowest level and work your way up. Before you can diagnose this problem, however, more information is required. Specifically, the candidate should ask these questions:

- What is the IP configuration of this subnet?
- What IP address, subnet mask, gateway and DNS address should be assigned to this workstation?
- What is the physical topology of the network?
- Is any other station on the network experiencing this problem?

This final question is the key because you can spend all day troubleshooting a workstation, when in reality the problem lies elsewhere on the network.

Assuming the problem has been confined to the workstation itself, the basic troubleshooting can begin. Here are some example questions that should be asked by the candidate as they go through the troubleshooting process.

Is the physical cable connected to the network card? Is there a link light on the network card? Is there a corresponding link light on the port to which the cable is connected (on a switch or a hub)? Does the operating system recognize the network card (On Linux, the 'lspci' command will list all devices connected to the PCI bus. On Solaris, use the command 'prtconf -v').

Are the IP configuration settings correct? What does the command 'ifconfig -a' tell you? Is any Layer 2 traffic being seen (the command 'arp -a' will display the Layer 2 address resolution protocol table)?

Is any host-based firewall in effect that could be blocking pings? Is the system that you are trying to ping available and not blocking pings? Can you ping other hosts on the network?

As you can see, there are many possible solutions to this problem. A good candidate will understand the methodical steps involved in solving a problem of this nature, and will implement those steps in the most logical order.

1.153. Your production UNIX system has been running fine when users suddenly complain that the system is slow. You log in and the system feels 'sluggish' and the terminal is unable to keep up

with your typing. What steps do you take to diagnose the problem?

The first step is to determine what is taking all of the system resources. Is a process consuming all of the CPU? Run the 'top' command to see if any processes are consuming the CPU cycles. Also check the load average with 'uptime' to see if the problem existed previously and just went away.

If no CPU load is evident, check the memory usage. Again, the 'top' command will display current memory and swap space in use. If all of the main memory is in use, and the system has to swap to disk for every new process, the response will decrease dramatically.

The final thing to check is disk usage. The 'vmstat' or 'sar -g' commands will display the disk usage. High page-out values mean the system is swapping because of low memory. High disk activity without paging is a sign of some processes doing extensive disk reads or writes.

This situation brings up an important point: it is impossible to know when a system is misbehaving if you don't know what it looks like when it is behaving. Establishing a system benchmark is critical to performance troubleshooting. It could be as simple as knowing what processes should show up in a 'ps -aef' output. A good system administrator will be able to answer these questions about his systems:

- What is the average number of running processes?
- What is the average number of users?
- What is the average CPU load?
- What is the average memory and swap usage?
- What is the average disk usage?

Without this information, diagnosing a problem situation becomes very difficult. There are many good monitoring tools available that can help in tracking this information. Of course, the standard UNIX monitoring tools run from a cron job can provide this information as well.

1.154. Describe the steps the system goes through, at boot time, before the users may sign on.

When a UNIX system is turned on, regardless of the architecture, a bootstrap program allows the administrator to boot the default UNIX kernel or specify the name of an old kernel if an older version is desired. The system will display information, among which the hardware recognition (when you can see if your parallel port is recognized) is the most important. Next, it verifies if the root filesystem needs checking (is "dirty" or was cleanly unmounted at shutdown). If the root filesystem needs checking, you will be prompted to allow the checking and repair (fsck utility) of the root filesystem. When the cleaning is complete, various scripts continue bringing up UNIX into either single or multiuser mode depending upon the configuration and options passed to the boot manager. Single-user mode is chosen mostly when system maintenance is needed. Once multiuser mode is chosen (or if it is the default boot level), a prompt is presented to enter the system time or take the default. The system then executes commands found in the /etc/rc* directories, generating startup messages for the various system services, such as the printer or network services. Finally, the system displays the login prompt.

1.155. When, and depending on what, are the other filesystems checked and mounted?

The other filesystems are mounted after the root filesystem mount is complete. They are checked and mounted depending on the entries found by the system in a mount table file at boot time (the name, location, and structure of this table vary between the UNIX vendors).

1.156. Name two ways to bring the system down.

The most commonly used, and the recommended way, is shutdown. This will warn the user that the system is coming down and unless otherwise specified, will have a one-minute grace period (may be different from one implementation to another). Simply turning the power off will not properly unmount the filesystem; instead, it could possibly force a lengthy check upon boot.

The second way is the haltsys (or halt for BSD) command. This command halts the system immediately. It should not be used unless in single-user mode. The users are logged out (their work will be lost) and network

servers and other programs are terminated abnormally. It is also a defined behavior that the system can be brought down by killing the init process with the default SIGINT signal, although this is rarely done in practice.

1.157. What are the files checked or scripts executed when a user logs in?

The order is: /etc/passwd, /etc/profile, $HOME/.profile, and in case the Korn shell is used, $HOME /. kshrc. If the C shell is used, .profile is replaced by .login and .kshrc by .cshrc (although the C shell and its derivatives are rapidly losing popularity).

1.158. When an account, for various reasons, is not used anymore, how can the system administrator disable it?

This can be accomplished in several ways. Assuming normal UNIX authentication, the administrator can change the password with the passwd command, alternately edit the /etc/passwd file and replace the second field of the users password entry with an asterix (*), or remove the entry from /etc/passwd entirely. If this is done, the entries should be removed from /etc/group and /etc/shadow as well, the home directory and mail spool of the user should be removed, and all files owned by the user should be deleted or assigned to other users. The administrator should run the find utility to ensure that no files remain on the system that are owned by the user. If something other than standard UNIX authentication is in use, such as NIS, DCE, Hesiod Bind, or a PAM-based authentication mechanism, this process will be different.

1.159. How can a user ensure that certain sensitive files cannot be read by others?

Use GnuPG (available at http://www.gnupg.org). To encrypt a plain text file with Gnupg, enter “gpg -c textfile” where “textfile” is the name of the file that you want to encrypt. Enter the password twice as requested, and “textfile.gpg” will appear in the same location as the original file. Delete the original file (if desired), and when the time comes to decrypt, simply enter “gpg textfile.gpg” and provide the valid passphrase. The original file should appear. The old UNIX crypt command should no longer be used. The contents of the files can be verified with cksum or md5sum if they are

available on your UNIX implementation and the checksums were previously computed.

1.160. How can a user remind another user about an important meeting taking place on a certain date, at a certain hour?

A very good way is to use the at command as it follows:

$ at 9:30am Mar 29
echo “Meeting at 10” | mail frank
<Ctrl-d>

1.161. How can a user execute a long command and still have the terminal available for work?

The user may execute the long command in the background using the & command as follows:

$ wc -c hugefil > wordcount &

1.162. Once the background command has been started, can it be aborted like any other foreground command?

If the shell has job control, the command “fg” will bring the background process to the foreground, where the interrupt key (Control-C) will kill it (unless a signal handler is installed). Otherwise, background commands cannot be aborted with the interrupt key; instead, you have to use the kill command. The shell will print a number when launching a background process. This number is the Process ID (PID) of the background process and would be supplied to the kill command to terminate the process. For example, kill 123 would send the default kill signal (SIGINT). It is possible that the program has configured itself to ignore SIGINT, in which case signal level 9 (SIGKILL) could be used to forcibly terminate the process, as in kill -9 123.

1.163. What is a filesystem?

A filesystem is a distinct division of the operating system consisting of files, directories, and the information needed to locate them. A filesystem

can be thought of as a structure upon which directories and files are constructed.

1.164. What happens if you have a directory such as /tmp, and you mount a filesystem such as /dev/u, on this directory?

If any files existed in the /tmp directory before the filesystem was mounted, they will disappear and then reappear when the filesystem is unmounted.

1.165. Can the system administrator enable users to mount a filesystem?

Normally, only the super user can mount or unmount filesystems. Some vendors offer proprietary UNIX extensions to grant this ability to non-root users, but a method to accomplish this that is largely portable across different UNIX versions is the use of the sudo package. A non-root user might need these permissions in order to mount a CD-ROM or a floppy.

1.166. Does it make sense for a directory to have the execute permission on it?

Yes, it does. If a directory does not have the execute permission on it, you cannot do a cd command to it.

1.167. By default, the GID (group identifier) of a newly created file is set to the GID of the creating process or user. How can this behavior be changed?

This behavior can be changed by setting the SGID bit on that directory which results in a new file having the GID of the directory.

1.168. What is a link between files, and what is the command you would use to link files?

A link is a directory entry referring to an inode. The same inode (file) can have several links. Any changes made to the file are effectively independent of the name by which the file is known. A file is not deleted until its last link is removed. The command to create links is ln.

1.169. What are the two types of links?

Hard links, which increase the link count for a file and cannot span filesystem boundaries, and soft links, which can span filesystems but do not increase the link count.

1.170. Which type of link consumes an inode, and which type does not?

A soft link will always consume an inode, but secondary hard links will not. Each physical file in a filesystem consumes a single inode, regardless of the number of hard links. A soft link, which cannot contain inode information since it may be on a separate filesystem, must consume an inode.

1.171. What is the most common way of locating files?

The most common way is by using the find command, which enables you to locate all files with a specified name, permissions setting, size, type, owner, or last access or modification date. The locate and whereis commands are also prevalent on many versions of UNIX, and might be useful.

1.172. How can you search a file for occurrences of a word or phrase?

The grep command displays all lines in a file that contain the key word or phrase. It is a good practice to enclose the search pattern in single quotes in order to protect it from command substitution in the shell.

1.173. How can you list the names of all files in all subdirectories in which there is a certain pattern?

You can use a combination of two commands:

find . -exec grep -1 “pattern” {} \;

1.174. Assuming you have a big file and you want to clear it (empty it), but you want to keep the file and its permissions, what could you do?

You may use the following commands:

$ > filename #(for the Bourne family of shells)

Or

$ cat /dev/null > filename #(for C shell)

1.175. Assuming that a directory becomes too large (has too many entries) and you decide to make two directories out of it (one of which has the same name), how would you proceed?

On some UNIX filesystem implementations, a directory, even if you remove all files it contains, never shrinks. Therefore, you should make two new directories, move all files into them, remove the old directory, and then rename one of the new directories using the name of the old one.

1.176. What is the purpose of the lost+found directory in a filesystem?

During fsck (filesystem check) command, any files that are unreferenced but valid are placed in this special directory called lost+found. Depending upon the filesystem implementation, the fsck command does not create or extend this directory; therefore, it has to contain a sufficient number of empty slots for fsck to use when reconnecting files.

1.177. List three commands most often used for archive or data transfer between devices.

The three most common are: tar, cpio, and dd.

If you were to make a script to back up your system, how would you make it? The script would look something like:

cd /
find . -follow -print ! cpio -ovaBL > /dev/rmt0h 2> /u/list
where in the file /u/list a list of the backed-up files would be obtained. Otherwise, you could use an alternate approach of:
tar cvf /dev/rctO2 .

If you are using GNU tar, you can use the “cvzf” option to enable compression (but cpio has better ability to recover from faulty tape). These

scripts should have root's permissions.

1.178. Once you obtain a stable system, that is, a good kernel, what are some of the first things the system administrator should do?

First, a backup of the entire system should be made. Then, depending upon directions for the specific UNIX implementation, the administrator should make a bootable diskette and then a root filesystem diskette. This way, if the root filesystem gets corrupted beyond repair, you can boot the system and restore the system from tapes.

1.179. What steps should be followed when you find you have an unusable root filesystem?

In case of an unusable root filesystem, the administrator can either reinstall the OS, or, if good backups exist, run fsck on the root filesystem until it passes (or reformat it), then follow these steps:
Boot from the diskette.
Insert the root filesystem diskette, when prompted.
mount /dev/hdOroot /mnt
cd /mnt
./usr/bin/tar xvf ./dev/rct0
umount /dev/hdOroot
Reboot.

1.180. Can you copy files to a DOS floppy on a UNIX system?

Yes, the command to use is mcopy which is part of the "mtools" (available from http://www.tux.org/pub/tux/knaff/mtools/index.html). This command permits copying in both directions. An example is:

mcopy /john/send a:

1.181. How can you find out the baud rate, the parity scheme, and other information about a serial line?

By using the following command:
$ stty < /dev/ttySO

1.182. How do you set the terminal type?

The preferable method for setting the terminal type is to assign the type to the TERM variable. This is usually done in .profile or login sign-on scripts.

1.183. What performance tools can you use to diagnose system inefficiency?

Depending upon what tools are available in your UNIX version, top, sar, prof, time, timex, and also ps commands may be used. The top, ps, and sar commands view a "snapshot" image of system utilization. The time and timex commands gather detailed information on elapsed time and processor consumption over the life of a process. The prof tools enable the user to access detailed data on execution times for the individual functions within a program.

1.184. How can the time command be used?

This command reports the time and resources consumed during the execution of a command.

1.185. What is swapping (paging)?

As processes request more memory in order to run, the paging daemon is responsible for freeing up memory by writing pages of memory from other (hopefully inactive) processes to the disk swap area.

1.186. What should you do in the case of an intense swapping/paging activity?

This activity is a sign that not enough memory is available for applications. The best thing to do is to increase the memory size. A temporary solution might be to decrease the buffer cache size, although this might decrease disk performance.

1.187. How can you check the status of processes?

You can check the status of your task using ps command. This command lists all the active processes that are running, both in the foreground and background. The most useful options for ps are ps -ef, while under BSD-style it is ps aux.

ps -ef
pd aux

1.188. How can you stop a runaway process?

To stop a process, use the kill command followed by an optional signal level and the required process ID. Signal names, their levels, and their normal behavior differ between UNIX distributions. Omitting the optional signal level normally passes signal 2 (SIGINT) to the process.

kill -9 pid

1.189. Assuming that you have to start the execution of a long command and you want to log out and go home, how do you go about it?

You can start the command in background, using the nohup command. The command would look like this: nohup command &. This traps the HUP signal that is sent when you exit the shell. You can now log out, and the process continues in background.

nohup command &

1.190. How can you prioritize processes?

Assuming you do not need the results of your command immediately, use the nice command, for example, nice -15 latecommand. The command will execute with the priority of 35, instead of the default of 20. Only the root user can increase the priority of a process; other users can only affect a decrease. The root user, and only the root user, is allowed to raise the priority of a process which can be accomplished with a command of the form nice +5 earlycommand. Some versions of UNIX also include the renice command, which can be used to adjust the priority of a process already running.

1.191. How can you schedule programs to run at specific times?

UNIX systems enable you to run programs automatically at specified times by using the cron or at programs. For cron, first create a crontab file, and then submit it to cron by using the crontab command.

1.192. What is a shell?

A shell is a command interpreter. The shell gives the user a high-level language in which to communicate with the operating system.

1.193. How does the shell execute the commands?

For the shell to execute a command, it has to be executable (to have the execute permission). If the command is a compiled program, the shell, as parent, creates a child process that immediately executes that program. If the command is a shell procedure (file containing shell commands), the shell forks a child to read the file and execute the commands inside.

1.194. How does the shell locate commands that the user wishes to execute?

The sequence of directories that are searched is given by the shell PATH variable. Directory pathnames are separated by colons.

1.195. Which are the characters you could use to form regular expressions to match other characters?

Some of these special characters are: the asterisk (*), which matches any string, including the null string; the question mark (?), which matches any one character; and any sequence of characters enclosed within brackets ([and]), which matches any one of the enclosed characters. Extended regular expressions and regular expressions unique to the PERL scripting language add much more flexibility to these simple wildcard functions.

1.196. How can you redirect the output of a command?

To redirect the output of a command, use > or >> redirection arguments. When > argument is used, a file will be created (or replaced if it exists) as standard output. When >> argument is used, the standard output is appended to the end of the file. The error output can also be redirected, but the semantics of doing so differ depending upon the shell family in use.

1.197. How can a command take the input from a file?

Make a file the input for a command by using the < argument. For instance: sort < infile.

1.198. What is command substitution?

Any command line can be placed within back quotation marks (‘) so that the output of the command replaces the quoted command itself. This concept is known as command substitution.

1.199. What are positional parameters (arguments)?

When a shell procedure is invoked, the shell implicitly creates positional parameters (arguments). The name of the shell procedure itself in position zero on the command line is assigned to the positional parameter $0. The first command argument is called $1, and so on.

1.200. What is a pipeline for a shell script?

A pipeline is a sequence of one or more commands separated by vertical bars (|). In a pipeline, the standard output of each command (except the last) is connected (by a pipe) to the standard input of the next command. Each command in a pipeline is run separately; the shell waits for the last command to finish. A simple example of a pipeline is

$cat /etc/passwd | grep luser | awk -F: `{print $3}’ | more

1.201. How can you use parentheses or braces at the command line or in a shell procedure?

A simple command in a pipeline can be replaced by a command list enclosed in either parentheses or braces. The output of all the enclosed commands is combined into one stream that becomes the input to the next command in the pipeline.

1.202. What is the difference between using parentheses and using braces?

When parentheses are used, the shell forks a subshell that reads and executes the enclosed commands. Unlike parentheses, no subshell is forked for braces; the enclosed commands are simply read and executed by the shell.

1.203. What are functions from the shell’s standpoint?

Functions are like separate shell scripts, except they reside in memory and are executed by the shell process, not by a separate process.

1.204. What is a command's environment?

All variables and their associated values that are known to a command at the beginning of its execution make up its environment. This environment includes variables that the command inherits from its parent process and variables specified as keyword parameters on the command line that invoke the command. The variables that a shell passes to its child processes are those that have been named as arguments to the export command (or the setenv command for the C shell).

1.205. How can you initialize new variables in a shell script and use them later in the sign-on shell?

In the Bourne shell family, use the dot (.) command, which causes the shell to read commands from the script without spawning a new sub-shell. In the C shell, the source command is used instead. Changes made to variables in the script are in effect after the dot command finishes. The command would look like this: .procedure (Bourne shell) or source procedure (C shell).

1.206. How can you solve the following situation: You want the users, once signed on, to be put in the application program; once they quit the application, you want them signed off the system?

A very efficient way is to put in their .profile or .login script, as the last command, the exec command, with the name of the application as an argument. In this way, once the users quit the application, they are signed off the system.

Another way is to set the application program as the login shell in /etc/passwd.

1.207. How can you make sure that the work file created during the execution of a script will be removed even if the script was interrupted?

Within a shell script, you can still take action (in this case remove files) when an interrupt signal is received by using the trap command. An

example of a trap command is

trap ‘rm -f tempfile; exit’ 0 2 3 15

1.208. Assuming you have to run backups during the night, and there is no night operator and the backup utility prompts you for answers before it starts, how would you solve this problem?

Create a shell script and put it in the crontabs directory, to be run by the cron process. The script would use a “here” document: it would have the interactive command (command which requires answers) followed by << eoi, where eoi is an arbitrary string, signaling the end of input. If the command uses the curses library, a more complicated solution would be required, possibly involving the expect utility. An example is
command <<- eoi

1.209. How can you write the standard output and the standard error for a command to the same file?

You can do this by using the file descriptors. A file descriptor is a handle to an open file. The C programming language defines default file pointers STDIN, STDOUT, and STDERR (standard input, output, and error). Pipelines and simple redirection alter STDIN and STDOUT, but STDERR can be explicitly directed in the Bourne family with the “2>“ notation (this feature is not available in the C shell, which is one of its great failings). If necessary, STDOUT can be explicitly referred to as “1>“ if desired. Furthermore, STDOUT and STDERR can be merged into the same file with the “2>&1” notation. An example command looks like this:

command 1>file name 2>&1

1.210. In a command list, how can you arrange the commands, so if one of them fails, the execution of the list stops?

The list would look something like this:
commandl && command2 && command3 &&...&& commandn

1.211. In a command list, how can you arrange the commands so that, the second one is executed only if the first one fails?

The list would look something like this:
commandl || command2
The first command is executed, its exit status is examined and only if it has a nonzero value, the second command is executed.

1.212. What can you do to fix the problem if one of your filesystems runs out of inodes?

You can back up the filesystem, unmount the filesystem, rerun mkfs and specify more inodes for the filesystem, mount the filesystem, and restore the backups.

1.213. How can you check the amount of free space on filesystems?

You can use the df command. df command reports filesystem disk space usage. On some systems, this utility reports sizes in 512 byte blocks.

1.214. What does it mean if a user has write permissions on a directory but is unable to remove files from that directory?

It means that the sticky bit has been set for that directory. The sticky bit is a directory protection setting that allows only the owner of the file (or superuser) to remove files from that directory.

1.215. What command can be used to compare two text files?

The diff command can be used to find out what lines must be changed in two files to bring them into agreement. This is especially useful for finding the differences between two versions of the same source program. The diff3 command is useful for comparing three separate files and merging the changes between them.

1.216. How can you convert a text file from UNIX format to MS-DOS format?

DOS text files differ from UNIX text files in that they contain a carriage return (“\r”) in addition to a line feed (“\n “) at the end of each line. While most UNIX vendors include proprietary utilities for accomplishing these conversions, modern versions of the UNIX awk utility can strip the carriage return if called with the following syntax: awk ‘{sub(/\r/,””); print;}’ dos.txt

> UNIX.txt. A file could be converted back with awk using a reverse syntax: awk ‘{sub(/$/,” \ r”); print;}’ UNIX.txt > dos.txt.

1.217. What is umask?

umask is a three-digit octal number used by the system to establish the permissions for a newly created file. The system subtracts this number from octal 777, and the results are the permissions for the new file. Plain files will never be created with the execute bit set (using an implicit mask), but directories allow the execute bit (directory scan privilege) to pass through.

1.218. How can you find out what variables are currently assigned?

By typing the command env, with no arguments, you can get a list of the variables currently assigned. The list can be lengthy, so it might be wise to pipe the output of env to a greater number.

1.219. What command can you use to archive a raw device?

You can use the dd command, which allows you to specify the input and output block size, among other options.

1.220. What are device drivers?

Device drivers are software routines, part of the kernel, which inter-face with the hardware. UNIX represents most devices as block- or character-special files located in the /dev directory. Each has a major and a minor device number.

1.221. What are UNIX daemons?

The UNIX daemons are a collection of processes that each perform a particular system task. Two examples are init and crond daemons.

1.222. Assuming you want to start the database engine every time you boot the system, how would you do it?

Add the necessary commands in /etc/rc* scripts, (scripts executed when the system is booting).

1.223. When the system is coming down, how would you bring down the database engine?

Locate the /etc/rc* scripts that perform system shutdown and add the entry there. If such entries cannot be found, add the necessary commands in the shutdown script (the path to the script varies among UNIX distributions), which is an old pathname for the shutdown command.

1.224. What can you do if a scrambled terminal responds to keyboard input but the display is incorrect?

First, check the TERM variable by entering env at the command line. If the terminal type is incorrect, reset it by entering at the command line TERM=wy60; export TERM (or setenv TERM wy60 for csh), if the terminal is a Wyse6O. After resetting the terminal type, reinitialize the terminal by entering tset with no arguments.

1.225. What can you do if a terminal that responds to keyboard input does not display the characters entered at the keyboard?

Sometimes, when a program stops prematurely as a result of an error, or when the user presses the <Break> key, the terminal stops echoing. To restore the terminal to normal operation, enter the following: <Ctrl>q<Ctrl>j stty sane <Ctrl>j. The <Ctrl>q unlocks any previous <Ctrl>s (which suspends output), and <Ctrl>j is the same as the enter/return key. The terminal should now display keyboard input.

1.226. How can all the files ending in ".foo" in a directory be renamed to end in ".bar"?

Assuming Bourne shell syntax, here is one way:
for x in *.foo; do base=‘basename $x .foo’; my $x $base.bar; done

1.227. Why wouldn't the administrator be able to unmount a filesystem?

No files can be open in the filesystem at unmount time. Additionally, all users must cd out of the filesystem. Until no processes are using data in the file system, it cannot be unmounted. The lsof (LiSt Open Files) can be helpful in such a situation, as can the kill command.

The following shells are installed on a UNIX system:
/bin/ash /bin/bash /bin/csh /bin/ksh /bin/pdksh
/bin/sash /bin/sh /bin/smrsh
/bin/tcsh /bin/zsh

1.228. Which of these shells will natively run Bourne shell scripts?

ash, bash, ksh, pdksh, sh will run scripts written for the Bourne shell. zsh may run these scripts with modifications to its environment. It is rare to see all of these shells installed on a UNIX system.

1.229. What function is used to delete a file within C?

The unlink function, with the single argument of the pathname to be deleted, that is, unlink("/home/luser/deleteme");.

1.230. What is the basic function used to get information about a file (permissions, owner, size, etc.)?

The stat() function and its derivatives return detailed information about a file in a stat structure. The stat structure contains the permissions, inode number, filesystem device number, device number for special files, number of links, uid of the owner, gid of the owner, size in bytes, times for last access, modification, file status change, best I/O block size, and the number of blocks allocated for the file.

1.231. What function is used to create a named pipe?

The mkfifo() function can be used to create a named pipe. This is a file to which processes can be connected to exchange information, which the following shell commands demonstrate:

```
$ mkfifo /tmp/fifo
$ echo -e `hello \n how are you \n goodbye' > /tmp/fifo & [1] 123
$ $ grep h /tmp/fifo
hello
how are you
```

In the above command, the "echo" was able to communicate with the "grep" through the named pipe (/tmp/fifo).

1.232. How can a C program set a timer to alert itself after a certain interval has passed?

A timer can be set with the alarm() function. The number of seconds until the alarm is desired is passed as an argument. When the time expires, the process is sent the SIGALRM signal.

1.233. How can a C program catch a signal? How can a Bourne shell script catch a signal?

The signal() function, which is part of ANSI C (not just UNIX C), enables an arbitrary function to be called when a desired signal is received. The Bourne shell trap command serves a similar purpose.

1.234. What information does the size command report about a command object file?

The size command reports the size of the text, data, and bss segments for a program.

1.235. When the fork() function splits one process into two, what is the return value for the parent, and what is the return value for the child?

Assuming success, the child receives zero, and the parent receives the PID of the child.

1.236. What is the difference between fork() and vfork()?

A call to fork() will make copies of the data, bss, stack, and heap segments of a program (the address space). If the child intends to replace itself with another process using an exec() function, the duplication of these data segments is both unnecessary and time-consuming. A call to vfork() puts the parent to sleep until the child has exited, and it causes the child to run in the same process space as the parent, negating the requirement to duplicate the address space.

1.237. What is a zombie process, and how is it avoided?

When a process exits, it is the responsibility of the parent to call one of the wait() functions to read the process return value. The dying child process gives up all its memory, but the kernel must maintain at least the PID, the termination status, and the amount of CPU time that the process consumed. If the parent does not call wait(), the kernel is forced to retain this information, and reflects this with "zombie" or "defunct" processes in the process list.

1.238. Other than signals, pipes, and files of any sort, what mechanisms are available to UNIX processes that enable them to exchange information?

Inter-Process Communication (IPC) under UNIX includes message queues, which allow small messages to be exchanged between processes; semaphores, which can be used to control access to a shared resource; and shared memory, which allow processes to exchange large blocks of data.

1.239. What is the difference between fopen() and open()?

The fopen() function is defined in ANSI standard C as a portable method to read and write files. It returns a "file pointer" which is supported on all C platforms, and makes a variety of functions available for performing highly formatted input and output (printf() and scanf()). The open() function returns a "file descriptor," which is a much lower-level protocol that is accessed with the (unformatted) read() and write() system calls.

1.240. What is mmap(), and why is it better than read() or write()?

The mmap() function maps a section of a file into memory, allowing the file to be accessed with pointer arithmetic (char *). Under certain derivatives of UNIX, this allows much faster access to the file because the kernel is not forced to copy the file's contents between multiple buffers.

1.241. How do I remove a file whose name begins with a "-" ?

Figure out some way to name the file so that it doesn't begin with a dash. The simplest answer is to use
rm ./-filename
(assuming "-filename" is in the current directory, of course.) This method of avoiding the interpretation of the "-" works with other commands too.

Many commands, particularly those that have been written to use the "getopt(3)" argument parsing routine, accept a "--" argument which means "this is the last option, anything after this is not an option", so your version of rm might handle "rm -- -filename". Some versions of rm that don't use getopt() treat a single "-" in the same way, so you can also try "rm - -filename".

1.242. How do I remove a file with funny characters in the filename?

If the 'funny character' is a '/', skip to the last part of this answer. If the funny character is something else, such as a ' ' or control character or character with the 8th bit set, keep reading.

The classic answers are

rm -i some*pattern*that*matches*only*the*file*you*want

which asks you whether you want to remove each file matching the indicated pattern; depending on your shell, this may not work if the filename has a character with the 8th bit set (the shell may strip that off);

and

rm -ri .

which asks you whether to remove each file in the directory. Answer "y" to the problem file and "n" to everything else. Unfortunately this doesn't work with many versions of rm. Also unfortunately, this will walk through every subdirectory of ".", so you might want to "chmod a-x" those directories temporarily to make them unsearchable.

Always take a deep breath and think about what you're doing and double check what you typed when you use rm's "-r" flag or a wildcard on the command line;

and

find . -type f ... -ok rm '{}' \;

where "..." is a group of predicates that uniquely identify the file. One possibility is to figure out the inode number of the problem file (use "ls -i .") and then use

find . -inum 12345 -ok rm '{}' \;
or
find . -inum 12345 -ok mv '{}' new-file-name \;

"-ok" is a safety check - it will prompt you for confirmation of the command it's about to execute. You can use "-exec" instead to avoid the prompting, if you want to live dangerously, or if you suspect that the filename may contain a funny character sequence that will mess up your screen when printed.

What if the filename has a '/' in it?

These files really are special cases, and can only be created by buggy kernel code (typically by implementations of NFS that don't filter out illegal characters in file names from remote machines.) The first thing to do is to try to understand exactly why this problem is so strange.

Recall that Unix directories are simply pairs of filenames and inode numbers. A directory essentially contains information like this:

filename inode

file1 12345
file2.c 12349
file3 12347

Theoretically, '/' and '\0' are the only two characters that cannot appear in a filename - '/' because it's used to separate directories and files, and '\0' because it terminates a filename.

Unfortunately some implementations of NFS will blithely create filenames with embedded slashes in response to requests from remote machines. For instance, this could happen when someone on a Mac or other non-Unix machine decides to create a remote NFS file on your Unix machine with the date in the filename. Your Unix directory then has this in it:

filename inode

91/02/07 12357

No amount of messing around with 'find' or 'rm' as described above will delete this file, since those utilities and all other Unix programs, are forced to interpret the '/' in the normal way.

Any ordinary program will eventually try to do unlink("91/02/07"), which as far as the kernel is concerned means "unlink the file 07 in the subdirectory 02 of directory 91", but that's not what we have - we have a *FILE* named "91/02/07" in the current directory. This is a subtle but crucial distinction.

What can you do in this case? The first thing to try is to return to the Mac that created this crummy entry, and see if you can convince it and your local NFS daemon to rename the file to something without slashes.

If that doesn't work or isn't possible, you'll need help from your system manager, who will have to try the one of the following. Use "ls -i" to find the inode number of this bogus file, then unmount the file system and use "clri" to clear the inode, and "fsck" the file system with your fingers crossed. This destroys the information in the file. If you want to keep it, you can try:

create a new directory in the same parent directory as the one containing the bad file name;

move everything you can (i.e. everything but the file with the bad name) from the old directory to the new one;

do “ls -id” on the directory containing the file with the bad name to get its inumber;

umount the file system;

“clri” the directory containing the file with the bad name;

“fsck” the file system.

Then, to find the file, remount the file system; rename the directory you created to have the name of the old directory (since the old directory should have been blown away by “fsck”) move the file out of “lost+found” into the directory with a better name.

Alternatively, you can patch the directory the hard way by crawling around in the raw file system. Use “fsdb”, if you have it.

1.243. How do I get a recursive directory listing?

One of the following may do what you want:

```
ls -R                   (not all versions of “ls” have -R)
find . -print           (should work everywhere)
du -a .                 (shows you both the name and size)
```

If you’re looking for a wildcard pattern that will match all “.c” files in this directory and below, you won’t find one, but you can use

```
% some-command `find . -name ‘*.c’ -print`
```

“find” is a powerful program.

1.244. How do I get the current directory into my prompt?

It depends which shell you are using. It’s easy with some shells, hard or impossible with others.

C Shell (csh):
Put this in your .cshrc - customize the prompt variable the way you want.

alias setprompt ‘set prompt=“${cwd}% “‘
setprompt # to set the initial prompt
alias cd ‘chdir \!* && setprompt’

If you use pushd and popd, you’ll also need

alias pushd ‘pushd \!* && setprompt’
alias popd ‘popd \!* && setprompt’

Some C shells don’t keep a $cwd variable - you can use `pwd` instead.

If you just want the last component of the current directory in your prompt (“mail% “ instead of “/usr/spool/mail% “) you can use

alias setprompt ‘set prompt=“$cwd:t% “‘

Some older csh’s get the meaning of && and || reversed. Try doing:

false && echo bug

If it prints “bug”, you need to switch && and || (and get a better version of csh.)

Bourne Shell (sh):

If you have a newer version of the Bourne Shell (SVR2 or newer) you can use a shell function to make your own command, “xcd” say:

```
xcd() { cd $* ; PS1=“`pwd` $ “; }
```

If you have an older Bourne shell, it’s complicated but not impossible. Here’s one way. Add this to your .profile file:

LOGIN_SHELL=$$ export LOGIN_SHELL
CMDFILE=/tmp/cd.$$ export CMDFILE

```
# 16 is SIGURG, pick a signal that's not likely to be used
PROMPTSIG=16 export PROMPTSIG
trap '. $CMDFILE' $PROMPTSIG
```

and then put this executable script (without the indentation!), let's call it "xcd", somewhere in your PATH

```
: xcd directory - change directory and set prompt
: by signalling the login shell to read a command file
cat >${CMDFILE?"not set"} <<EOF
cd $1
PS1="\`pwd\`$ "
EOF
kill -${PROMPTSIG?"not set"} ${LOGIN_SHELL?"not set"}
```

Now change directories with "xcd /some/dir".

Korn Shell (ksh):

Put this in your .profile file:

```
PS1='$PWD $ '
```

If you just want the last component of the directory, use

```
PS1='${PWD##*/} $ '
```

T C shell (tcsh)

Tcsh is a popular enhanced version of csh with some extra builtin variables (and many other features):

%~	the current directory, using ~ for $HOME
%d or %/	the full pathname of the current directory
%c or %.	the trailing component of the current directory

so you can do

```
set prompt='%~ '
```

BASH (FSF's "Bourne Again SHell")

\w in $PS1 gives the full pathname of the current directory, with ~ expansion for $HOME; \W gives the basename of the current directory. So, in addition to the above sh and ksh solutions, you could use

```
PS1='\w $ '
or
PS1='\W $ '
```

1.245. How do I read characters from the terminal in a shell script?

In sh, use read. It is most common to use a loop like

```
while read line
   do
            ...
   done
```

In csh, use $< like this:

```
while ( 1 )
        set line = "$<"
        if ( "$line" == "" ) break
        ...
   end
```

Unfortunately csh has no way of distinguishing between a blank line and an end-of-file.

If you're using sh and want to read a *single* character from the terminal, you can try something like

```
echo -n "Enter a character: "
stty cbreak                     # or  stty raw
readchar=`dd if=/dev/tty bs=1 count=1 2>/dev/null`
stty -cbreak
```

echo “Thank you for typing a $readchar .”

1.246. How do I rename “*.foo” to “*.bar”, or change file names to lowercase?

Why doesn’t “mv *.foo *.bar” work? Think about how the shell expands wildcards. “*.foo” and “*.bar” are expanded before the mv command ever sees the arguments. Depending on your shell, this can fail in a couple of ways. CSH prints “No match.” because it can’t match “*.bar”. SH executes “mv a.foo b.foo c.foo *.bar”, which will only succeed if you happen to have a single directory named “*.bar”, which is very unlikely and almost certainly not what you had in mind.

Depending on your shell, you can do it with a loop to “mv” each file individually. If your system has “basename”, you can use:

C Shell:

```
foreach f ( *.foo )
  set base=`basename $f .foo`
  mv $f $base.bar
end
```

Bourne Shell:

```
for f in *.foo; do
  base=`basename $f .foo`
  mv $f $base.bar
done
```

Some shells have their own variable substitution features, so instead of using “basename”, you can use simpler loops like:

C Shell:

```
foreach f ( *.foo )
  mv $f $f:r.bar
end
```

Korn Shell:

```
for f in *.foo; do
   mv $f ${f%foo}bar
done
```

If you don't have "basename" or want to do something like renaming foo.* to bar.*, you can use something like "sed" to strip apart the original file name in other ways, but the general looping idea is the same. You can also convert file names into "mv" commands with 'sed', and hand the commands off to "sh" for execution. Try

```
ls -d *.foo | sed -e 's/.*/mv & &/' -e 's/foo$/bar/' | sh
```

A program by Vladimir Lanin called "mmv" that does this job nicely was posted to comp.sources.unix (Volume 21, issues 87 and 88) in April 1990. It lets you use

```
mmv '*.foo' '=1.bar'
```

Shell loops like the above can also be used to translate file names from upper to lower case or vice versa. You could use something like this to rename uppercase files to lowercase:

C Shell:

```
foreach f ( * )
        mv $f `echo $f | tr '[A-Z]' '[a-z]'`
end
```

Bourne Shell:

```
for f in *; do
        mv $f `echo $f | tr '[A-Z]' '[a-z]'`
done
```

Korn Shell:

```
typeset -l l
for f in *; do
        l="$f"
        mv $f $l
```

```
  done
```

If you wanted to be really thorough and handle files with `funny' names (embedded blanks or whatever) you'd need to use

Bourne Shell:

```
  for f in *; do
    g=`expr "xxx$f" : 'xxx\(.*\)' | tr '[A-Z]' '[a-z]'`
    mv "$f" "$g"
  done
```

The `expr' command will always print the filename, even if it equals `-n' or if it contains a System V escape sequence like `\c'.

Some versions of "tr" require the [and], some don't. It happens to be harmless to include them in this particular example; versions of tr that don't want the [] will conveniently think they are supposed to translate '[' to '[' and ']' to ']'. If you have the "perl" language installed, you may find this rename script by Larry Wall very useful. It can be used to accomplish a wide variety of filename changes.

```
#!/usr/bin/perl
#
# rename script examples from lwall:
#       rename 's/\.orig$//' *.orig
#       rename 'y/A-Z/a-z/ unless /^Make/' *
#       rename '$_ .= ".bad"' *.f
#       rename 'print "$_: "; s/foo/bar/ if $_ =~ /^y/i' *

$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die $@ if $@;
    rename($was,$_) unless $was eq $_;
}
```

1.247. Why do I get [some strange error message] when I “rsh host command” ?

We’re talking about the remote shell program “rsh” or sometimes “remsh” or “remote”; on some machines, there is a restricted shell called “rsh”, which is a different thing.

If your remote account uses the C shell, the remote host will fire up a C shell to execute ‘command’ for you, and that shell will read your remote .cshrc file. Perhaps your .cshrc contains a “stty”, “biff” or some other command that isn’t appropriate for a non-interactive shell. The unexpected output or error message from these commands can screw up your rsh in odd ways.

Here’s an example. Suppose you have

```
stty erase ^H
biff y
```

in your .cshrc file. You’ll get some odd messages like this.

```
% rsh some-machine date
stty: : Can’t assign requested address
Where are you?
Tue Oct  1 09:24:45 EST 1991
```

You might also get similar errors when running certain “at” or “cron” jobs that also read your .cshrc file.

Fortunately, the fix is simple. There are, quite possibly, a whole *bunch* of operations in your “.cshrc” (e.g., “set history=N”) that are simply not worth doing except in interactive shells. What you do is surround them in your “.cshrc” with:

```
if ( $?prompt ) then
        operations....
endif
```

and, since in a non-interactive shell "prompt" won't be set, the operations in question will only be done in interactive shells. You may also wish to move some commands to your .login file; if those commands only need to be done when a login session starts up (checking for new mail, unread news and so on) it's better to have them in the .login file.

1.248. How do I {set an environment variable, change directory} inside a program or shell script and have that change affect my current shell?

In general, you can't, at least not without making special arrangements. When a child process is created, it inherits a copy of its parent's variables (and current directory). The child can change these values all it wants but the changes won't affect the parent shell, since the child is changing a copy of the original data.

Some special arrangements are possible. Your child process could write out the changed variables, if the parent was prepared to read the output and interpret it as commands to set its own variables.

Also, shells can arrange to run other shell scripts in the context of the current shell, rather than in a child process, so that changes will affect the original shell.

For instance, if you have a C shell script named "myscript":

```
cd /very/long/path
setenv PATH /something:/something-else
```

or the equivalent Bourne or Korn shell script

```
cd /very/long/path
PATH=/something:/something-else export PATH
```

and try to run "myscript" from your shell, your shell will fork and run the shell script in a subprocess. The subprocess is also running the shell; when it sees the "cd" command it changes *its* current directory, and when it

sees the “setenv” command it changes *its* environment, but neither has any effect on the current directory of the shell at which you’re typing (your login shell, let’s say).

In order to get your login shell to execute the script (without forking) you have to use the “.” command (for the Bourne or Korn shells) or the “source” command (for the C shell). I.e. you type

. myscript

to the Bourne or Korn shells, or

source myscript

to the C shell.

If all you are trying to do is change directory or set an environment variable, it will probably be simpler to use a C shell alias or Bourne/Korn shell function. See the “how do I get the current directory into my prompt” section of this article for some examples.

1.249. How do I redirect stdout and stderr separately in csh?

In csh, you can redirect stdout with “>“, or stdout and stderr together with “>&” but there is no direct way to redirect stderr only. The best you can do is

(command >stdout_file) >&stderr_file

which runs “command” in a subshell; stdout is redirected inside the subshell to stdout_file, and both stdout and stderr from the subshell are redirected to stderr_file, but by this point stdout has already been redirected so only stderr actually winds up in stderr_file.

If what you want is to avoid redirecting stdout at all, let sh do it for you.

sh -c ‘command 2>stderr_file’

1.250. How do I tell inside .cshrc if I'm a login shell?

From: msb@sq.com (Mark Brader)
Date: Mon, 26 Oct 1992 20:15:00 -0500

When people ask this, they usually mean either

How can I tell if it's an interactive shell? or
How can I tell if it's a top-level shell?

You could perhaps determine if your shell truly is a login shell (i.e. is going to source ".login" after it is done with ".cshrc") by fooling around with "ps" and "$$". Login shells generally have names that begin with a '-'. If you're really interested in the other two questions, here's one way you can organize your .cshrc to find out.

```
if (! $?CSHLEVEL) then
        #
        # This is a "top-level" shell,
        # perhaps a login shell, perhaps a shell started up by
        # 'rsh machine some-command'
        # This is where we should set PATH and anything else we
        # want to apply to every one of our shells.
        #
        setenv    CSHLEVEL      0
        set home = ~username      # just to be sure
            source ~/.env         # environment stuff we always
want
else
        #
        # This shell is a child of one of our other shells so
        # we don't need to set all the environment variables again.
        #
        set tmp = $CSHLEVEL
        @ tmp++
        setenv    CSHLEVEL      $tmp
endif
```

```
# Exit from .cshrc if not interactive, e.g. under rsh
if (! $?prompt) exit

# Here we could set the prompt or aliases that would be useful
# for interactive shells only.

source ~/.aliases
```

How do I construct a shell glob-pattern that matches all files except “.” and “..” ?

You’d think this would be easy.

`*` Matches all files that don’t begin with a “.”;

`.*` Matches all files that do begin with a “.”, but this includes the special entries “.” and “..”,
which often you don’t want;

`.[!.]*` (Newer shells only; some shells use a “^” instead of the “!”; POSIX shells must accept the “!”, but may accept a “^” as well; all portable applications shall not use an unquoted “^” immediately following the “[“)

Matches all files that begin with a “.” and are followed by a non-”.”; unfortunately this will miss “..foo”;

`.??*` Matches files that begin with a “.” and which are at least 3 characters long. This neatly avoids “.” and “..”, but also misses “.a” .

So to match all files except “.” and “..” safely you have to use 3 patterns (if you don’t have filenames like “.a” you can leave out the first):

```
.[!.]* .??* *
```

Alternatively you could employ an external program or two and use backquote substitution. This is pretty good:

`ls -a | sed -e '/^\.$/d' -e '/^\.\.$/d'`

(or `ls -A` in some Unix versions)

but even it will mess up on files with newlines, IFS characters or wildcards in their names.

1.251. How do I find the last argument in a Bourne shell script?

If you are sure the number of arguments is at most 9, you can use:

```
eval last=\${$#}
```

In POSIX-compatible shells it works for ANY number of arguments. The following works always too:

```
for last
do
        :
done
```

This can be generalized as follows:

```
for i
do
        third_last=$second_last
        second_last=$last
        last=$i
done
```

Now suppose you want to REMOVE the last argument from the list, or REVERSE the argument list, or ACCESS the N-th argument directly, whatever N may be. Here is a basis of how to do it, using only built-in shell constructs, without creating subprocesses:

```
t0= u0= rest='1 2 3 4 5 6 7 8 9' argv=
for h in '' $rest
```

```
        do
                for t in "$t0" $rest
                do
                        for u in $u0 $rest
                        do
                                case $# in
                                0)
                                        break 3
                                esac
                                eval argv$h$t$u=\$1
                                argv="$argv \"\$argv$h$t$u\""           #
(1)
                                shift
                        done
                        u0=0
                done
                t0=0
        done
        # now restore the arguments
        eval set x "$argv"                                              # (2)
        shift
```

This example works for the first 999 arguments. Enough? Take a good look at the lines marked (1) and (2) and convince yourself that the original arguments are restored indeed, no matter what funny characters they contain!

To find the N-th argument now you can use this:

```
eval argN=\$argv$N
```

To reverse the arguments the line marked (1) must be changed to:

```
argv="\"\$argv$h$t$u\" $argv"
```

How to remove the last argument is left as an exercise.

If you allow subprocesses as well, possibly executing nonbuilt-in commands, the `argvN' variables can be set up more easily:

```
N=1
for i
do
        eval argv$N=\$i
        N=`expr $N + 1`
done
```

To reverse the arguments there is still a simpler method, that even does not create subprocesses. This approach can also be taken if you want to delete e.g. the last argument, but in that case you cannot refer directly to the N-th argument anymore, because the `argvN' variables are set up in reverse order:

```
argv=
for i
do
        eval argv$#=\$i
        argv="\"\$argv$#\" $argv"
        shift
done
eval set x "$argv"
shift
```

1.252. What's wrong with having '.' in your $PATH?

A bit of background: the PATH environment variable is a list of directories separated by colons. When you type a command name without giving an explicit path (e.g. you type "ls", rather than "/bin/ls") your shell searches each directory in the PATH list in order, looking for an executable file by that name, and the shell will run the first matching program it finds.

One of the directories in the PATH list can be the current directory "." . It is also permissible to use an empty directory name in the PATH list to indicate the current directory. Both of these are equivalent

for csh users:

setenv PATH :/usr/ucb:/bin:/usr/bin
setenv PATH .:/usr/ucb:/bin:/usr/bin

for sh or ksh users

PATH=:/usr/ucb:/bin:/usr/bin export PATH
PATH=.:/usr/ucb:/bin:/usr/bin export PATH

Having "." somewhere in the PATH is convenient - you can type "a.out" instead of "./a.out" to run programs in the current directory. But there's a catch.

Consider what happens in the case where "." is the first entry in the PATH. Suppose your current directory is a publically-writable one, such as "/tmp". If there just happens to be a program named "/tmp/ls" left there by some other user, and you type "ls" (intending, of course, to run the normal "/bin/ls" program), your shell will instead run "./ls", the other user's program. Needless to say, the results of running an unknown program like this might surprise you.

It's slightly better to have "." at the end of the PATH:

setenv PATH /usr/ucb:/bin:/usr/bin:.

Now if you're in /tmp and you type "ls", the shell will search /usr/ucb, /bin and /usr/bin for a program named "ls" before it gets around to looking in ".", and there is less risk of inadvertently running some other user's "ls" program. This isn't 100% secure though - if you're a clumsy typist and someday type "sl -l" instead of "ls -l", you run the risk of running "./sl", if there is one. Some "clever" programmer could anticipate common typing mistakes and leave programs by those names scattered throughout public directories. Beware. Many seasoned UNIX users get by just fine without having "." in the PATH at all:

setenv PATH /usr/ucb:/bin:/usr/bin

If you do this, you'll need to type "./program" instead of "program" to run programs in the current directory, but the increase in security is probably worth it.

1.253. What is the significance of the "tee" command?

It reads the standard input and sends it to the standard output while redirecting a copy of what it has read to the file specified by the user.

1.254. What does the command "$who | sort –logfile > newfile" do?

The input from a pipe can be combined with the input from a file. The trick is to use the special symbol "-" (a hyphen) for those commands that recognize the hyphen as std input. In the above command the output from who becomes the std input to sort, meanwhile sort opens the file logfile, the contents of this file is sorted together with the output of who (rep by the hyphen) and the sorted output is redirected to the file newfile.

1.255. What does the command "$ls | wc –l > file1" do?

ls becomes the input to wc which counts the number of lines it receives as input and instead of displaying this count , the value is stored in file1.

1.256. Which of the following commands is not a filter (a) man, (b) cat, (c) pg, (d) head

man. A filter is a program which can receive a flow of data from std input, process (or filter) it and send the result to the std output.

1.257. How is the command "$cat file2" different from "$cat >file2" and >> redirection operators?

> is the output redirection operator when used it overwrites while >> operator appends into the file.

1.258. Explain the steps that a shell follows while processing a command.

After the command line is terminated by the key, the shell goes ahead with processing the command line in one or more passes. The sequence is well

defined and assumes the following order:

- Parsing: The shell first breaks up the command line into words, using spaces and the delimiters, unless quoted. All consecutive occurrences of a space or tab are replaced here with a single space.
- Variable evaluation: All words proceeded by a $ are evaluated as variables, unless quoted or escaped.
- Command substitution: Any command surrounded by backquotes is executed by the shell which then replaces the standard output of the command into the command line.
- Wild-card interpretation: The shell finally scans the command line for wild-cards (the characters *, ?, [,]). Any word containing a wild-card is replaced by a sorted list of filenames that match the pattern. The list of these filenames then forms the arguments to the command.
- PATH evaluation: It finally looks for the PATH variable to determine the sequence of directories it has to search in order to hunt for the command.

1.259. What is difference between cmp and diff commands?

cmp - compares two files byte by byte and displays the first mismatch
diff - tells the changes to be made to make the files identical

1.260. What is the use of ‘grep’ command?

‘grep’ is a pattern search command. It searches for the pattern, specified in the command line with appropriate option, in a file(s).
Syntax: grep
Example: grep 99mx mcafile

1.261. What is the difference between cat and more command?

cat displays file contents. If the file is large the contents scroll off the screen before we view it. So command 'more' is like a pager which displays the contents page by page.

1.262. Write a command to kill the last background job?

kill $!

1.263. Which command is used to delete all files in the current directory and all its sub-directories?

rm -r *

1.264. Write a command to display a file's contents in various formats?

$od -cbd file_name
c - character, b - binary (octal), d-decimal, od=Octal Dump.

1.265. What will the following command do?

$echo *
It is similar to 'ls' command and displays all the files in the current directory.

1.266. Is it possible to create a new file system in UNIX?

Yes, 'mkfs' is used to create a new file system.

1.267. Is it possible to restrict incoming message?

Yes, using the 'mesg' command.

1.268. What is the use of the command "ls -x chapter[1-5]"

ls stands for list; so it displays the list of the files that starts with 'chapter' with suffix '1' to '5', chapter1, chapter2, and so on.

1.269. Is 'du' a command? If so, what is its use?

Yes, it stands for 'disk usage'. With the help of this command you can find the disk capacity and free space of the disk.

1.270. Is it possible to count number of char, line in a file; if so, How?

Yes, wc-stands for word count.
wc -c for counting number of characters in a file.
wc -l for counting lines in a file.

1.271. Name the data structure used to maintain file identification?

‘inode’, each file has a separate inode and a unique inode number.

1.272. How many prompts are available in a UNIX system?

Two prompts, PS1 (Primary Prompt), PS2 (Secondary Prompt).

1.273. How does the kernel differentiate device files and ordinary files?

Kernel checks 'type' field in the file's inode structure.

1.274. How to switch to a super user status to gain privileges?

Use ‘su’ command. The system asks for password and when valid entry is made the user gains super user (admin) privileges.

1.275. What are shell variables?

Shell variables are special variables, a name-value pair created and maintained by the shell. Example: PATH, HOME, MAIL and TERM

1.276. What is redirection?

Directing the flow of data to the file or from the file for input or output. Example: ls > wc

1.277. How to terminate a process which is running and the specialty on command kill 0?

With the help of kill command we can terminate the process.
Syntax: kill pid
kill 0 - kills all processes in your system except the login shell.

1.278. What is a pipe and give an example?

A pipe is two or more commands separated by pipe char '|'. That tells the shell to arrange for the output of the preceding command to be passed as input to the following command.
Example: ls -l | pr
The output for a command ls is the standard input of pr.

When a sequence of commands are combined using pipe, then it is called pipeline.

1.279. Explain kill() and its possible return values.

There are four possible results from this call:
‘kill()’ returns 0. This implies that a process exists with the given PID, and the system would allow you to send signals to it. It is system-dependent whether the process could be a zombie.
‘kill()’ returns -1, ‘errno == ESRCH’ either no process exists with the given PID, or security enhancements are causing the system to deny its existence. (On some systems, the process could be a zombie.)
‘kill()’ returns -1, ‘errno == EPERM’ the system would not allow you to kill the specified process. This means that either the process exists (again, it could be a zombie) or draconian security enhancements are present (e.g. your process is not allowed to send signals to *anybody*).
‘kill()’ returns -1, with some other value of ‘errno’ you are in trouble! The most-used technique is to assume that success or failure with ‘EPERM’ implies that the process exists, and any other error implies that it doesn't.
An alternative exists, if you are writing specifically for a system (or all those systems) that provide a ‘/proc’ filesystem: checking for the existence of ‘/proc/PID’ may work.

1.280. What is relative path and absolute path?

Absolute path: Exact path from root directory.
Relative path: Relative to the current path.